University of Victoria

CSC 370:

Database Systems

Professors: Dr. Sean Chester and Mr. Yichun Zhao

Textbooks:

Database Systems: The Complete Book

Conceptual Database Design: An Entity-Relationship Approach

Algorithm Design and Applications

Notes By: Dominique Charlebois

Fall 2022

Last Updated: December 6, 2022

Contents

0 Introduction

Overview

This course provides an overview of a relational database management system (RDBMS), from modelling the world and designing a database to querying the data and optimising performance in a manner that maintains the critical ACID properties of a database. The course will include elements of software engineering, theoretical computer science, programming in the SQL language, and computer systems.

In this short prelude to the course, you will become familiar with:

- the expectations of the course, including key deliverables.
- the objectives of the course, including why we will focus on relational databases.
- what is a database and why we would use one.
- the high-level process of database design.

Module-Level Intended Learning Outcomes

Upon completion of this short introductory module, you can:

- when reflecting on the course structure and goals, determine whether CSC 370 fits into your academic plans this semester.
- when reflecting on this upcoming semester, plan out your workload appropriately to meet the deliverables and learning objectives of the course.
- when reflecting on the first two modules of this course, articulate accurately how both conceptual and logical design fit into the database design process.

0.1 Course Structure

Lesson Scope

This preliminary lesson describes the structure of the course, what the student and instructors can expect from each other, and how the lessons will be delivered. We will walk through Brightspace, GitHub, and Slack as the tools for this course.

Lesson Objectives

- When deciding if you want to take this course this semester, you can informatively take into account the lecture delivery methods, assessment style, and learning objectives of the course.
- When planning out your semester, you can accurately anticipate dates for deliverables in this course.

Delivery Method

This lesson will predominantly consist of a walk-through of Brightspace to see where to find the relevant components for this course, as well as some pertinent web links. There is no slide deck associated with this lesson.

Preparation

You are encouraged to read through the official course outline in advance, as I will focus only on the sections where there is likely to be the most uncertainty (e.g., lecture delivery style throughout the semester and assessments).

Important Information

- Slack
- GitHub Repository
- Database Systems - The Complete book by Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom
- Conceptual Database Design - An Entity-Relationship Approach by Carlo Batini, Stefano Ceri, and Shamkant B. Navathe

0.2 Course Overview

0.2.1 Database

A database is more than a text file that stores data. It needs to be durable, efficient, concurrent, and consistent. It needs to be durable because we can't lose data (often times it is illegal to lose data). It needs to be efficient because we need to access data within a reasonable time. It needs to allow multiple users to access the data at the same time. And finally, it need to be consistent without errors (i.e., the data cannot have incompatible entries).

0.2.2 Database Management System

A database management system (DBMS) is a set of modularised units that take care of different aspects of managing data. This may include executing queries, dealing with the storage layer, or deal with logging.

0.2.3 Database Programming

We will be using declarative programming to interact with a relational database. In the case of SQL, it can look similar to the following:

```
SELECT name, SUM(length)
FROM MovieExec, Movies
WHERE producerC# = cert#
GROUP BY name
HAVING MIN(year) < 1930;
```

0.2.4 Data Model

A *data model* is a notation for describing data or information. It is an abstraction of three components: the *structure* of data, the *operations* that can be performed on the data, and the *constraints* that apply to that data.

0.2.5 Relation (Database)

A relation is abstractly a two-dimensional table. Table 1 is an example of a proper two-dimensional table. Table 2 is an example of an improper two-dimensional table.

We can also perform *operations* on a relation. For example, we can filter Table 1 by Evergreens. We can apply *constraints* to prevent a table similar to Table 2. Enforcing the data to have a type (evergreen or deciduous) eliminates false entries into the relation.

- An *attribute* is a column within the table. In Table 1 the three attributes are Latin Name, English Name, and Type.

- A *schemata* is the definition of a relations structure, in our case it can be represented as **Trees(LatinName, EnglishName, Type)**, where Trees is the name of the relation and the terms within the parenthesis are the set of attributes.
- A *tuple* is represented by each row of the relation. Thus, here t_1 would be (Taxus baccata, English Yew, and Evergreen).
- The *domain* is the possible values that an attribute can take. An example would be $Type \in \{Deciduous, Evergreen\}$.

A final note is that a *Relational Model* is unordered. So **Trees(LatinName, EnglishName, Type)** and **Trees(Type, LatinName, EnglishName)** are theoretically identical.

Latin Name	**English Name**	**Type**
Taxus Baccata	English Yew	Evergreen
Acer Saccharum	Sugar Maple	Deciduous
Thuja Plicata	Western Red Cedar	Evergreen

Table 1: A relational table

12	Evergreen	false
Taxus Baccata	English Yew	Evergreen
Acer Saccharum	Sugar Maple	Deciduous
Thuja Plicata	Western Red Cedar	Evergreen

Table 2: Not a relational table

1 Conceptual Database Design

Overview

A conceptual schema is a high-level, abstract description of a database that focuses on what concepts are modeled as data and how they interact with each other. Because it is independent of any particular system or structure of data, the creation of a conceptual schema is often considered the first step in designing a database.

One of the ubiquitous modelling languages for conceptual schemata is an Entity Relationship Diagram (ERD). In this module, we learn how to design a conceptual model for a database methodologically, how to express that model as an ERD, and how to differentiate good design from poor.

Module-Level Intended Learning Outcomes

Upon completion of this short introductory module, you can:

- given a set of requirements for an information system, produce a good conceptual model as an ERD that conforms to those requirements.
- when asked, describe a concrete methodology for building a complete conceptual model from scratch as a series of primitive transformations.
- given a ERD, evaluate whether it adheres to the qualities of good conceptual design and, if not, propose transformations that address its shortcomings.

References

Batini, Ceri, and Navathe (1992) [1]. Conceptual Database Design: An Entity-Relationship Approach. The Benjamin/Cummings Publishing Company, Inc. Chapters 2, 3, and 6.

Garcia-Molina, Ullman, and Widom (2009) [2]. Database Systems: The Complete Book. Pearson Prentice Hall. 2nd Edition. Chapter 4.

1.1 Data Modeling Concepts

Lesson Scope

This lesson covers §1.1–1.2, 1.5, and 2.1–2.2 of Batini et al (1992) [1]. It will introduce a process of creating information systems and databases, with a focus on data modeling. It takes the view of modeling as the selection and deliberate non-selection of abstractions and provides a systematic framework for thinking about data abstractions.

Lesson Objectives

1. When asked, you can explain how conceptual design fits into an information system's life cycle.
2. Given a data abstraction, you can identify the abstraction mechanism and the properties of the abstraction.

Delivery Method

This lesson will primarily follow the corresponding slide deck, which itself follows the textbook quite closely, including the choice of examples.

Preparation

You are encouraged to read the sections of the textbook in advance, but this is not required to follow the lecture.

1.1.1 Information Systems Life Cycle

Information systems is a collection of activities that regulate the sharing and distribution of information and the storage of data that are relevant to the management of the enterprise. A database is a large collection of structured data in a computer system. And a database management systems (DBMS) is a software package for managing databases.

We must first determine the cost effectiveness of various design alternatives. Then we can understand the mission of the information system. We will need to determine the specification of the information system's structure (such as database design and the application design). Once we have completed this step we can start prototyping and implementing the information system. If needed you can return to the requirements collection and analysis step after prototyping. Lastly, you'll need to validate and test the information system and deal with the operation. Figure 1 illustrate the general flow of the process. We will focus on the design and implementation steps.

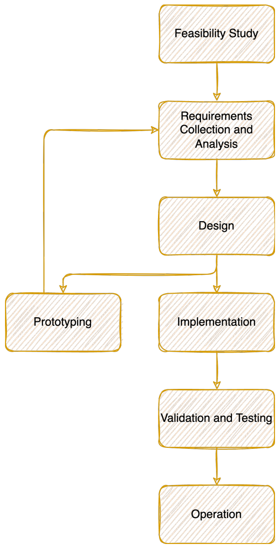


Figure 1: Information Systems Life Cycle

The life cycle is mostly a reference framework. However, the life cycle tells us that database design should be preceded by requirement analysis, should be conducted in parallel with application design, and should be followed by the implementation of either a prototype or a final system.

1.1.2 Phases of Database Design

We will be following a *data-driven* approach where you first design the database and then the application we can view this in Figure 2. We can also view this as a set of dependencies rather than a waterfall design. The data requirements is the outcome of the requirement analysis. Conceptual design describes the information content and is a high-level description of the structure of the database. It is *independent* of the class of database used for the implementation. Logical design is describing the information with respect to a specific class of database and the description of the structure of a database that could be processed by the DBMS software. And finally, Physical design is the storage structures and access methods, tailored to a specific DBMS.

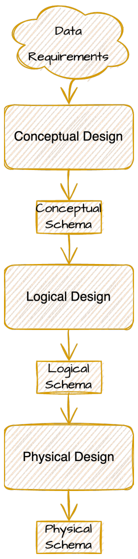


Figure 2: Data-Driven Approach to Information Systems Design

Conceptual Design Conceptual design starts from the specification of requirements and results in the conceptual schema of the database. A **conceptual model** is a language that is used to describe conceptual schemas.

Logic Design Logical design starts from the conceptual schema and results in the logical schema. A **logical schema** is a description of the structure of the database that can be processed by the DBMS software. A **logical model** is a language that is used to specify logical schemas; the most widely used logical models belong to three classes: relational, network, and hierarchical.

Physical Design Physical design starts from the logical schema and results in the physical schema. A **physical schema** is a description of the implementation of the database in secondary memory; it describes the storage structures and access methods used in order to effectively access data.

Table 3 shows the dependencies of conceptual, logical, and physical design. We note that each design step increases in dependency.

1.1.3 Conceptual Design

Conceptual design is difficult because it involves multiple stakeholders, it has no automated tools, and starts from natural language requirements. The product from the conceptual design is a conceptual schema. The conceptual design should be independent

Dependence of on:	DBMS Class	Specific DBMS
Conceptual Design	No	No
Logical Design	Yes	No
Physical Design	Yes	Yes

Table 3: Dependence of conceptual, logical, and physical design on the class of DBMS and the specific DBMS.

of the database class. It should be equally compatible with any choice. Table 3 shows us that it isn't dependent on anything.

"If we know the class of database that we want to use, conceptual design may seem redundant." However:

1. Subsequent phases can be partly automated, but a designer needs to convert requirements into an initial schema.
2. Success at this stage depends on cooperation with users; conceptual models do not require much technical expertise.
3. Choice of target DBMS can be postponed until after co-design.
4. If requirements change, conceptual design may be a better starting point of the new design activity.
5. Different databases, even those from different classes, can be compared in a homogeneous framework, which is particularly helpful when the data is heterogeneous and the information system uses more than one database.
6. Conceptual schema remains as easy-to-understand documentation.

1.1.4 Data Modeling

A *Data Model* is a vehicle for describing reality. A *Data Schema* is a representation of reality. And an *Abstraction* is a representative of characteristics and properties relevant to the data.

Thus, data modeling is a process of abstraction which uses a small collection of primitive abstraction. They can be seen as **Classification**, **Aggregation**, and **Generalisation**. We cannot say whether the abstraction is good or not because it truly depends on the requirements. An example of an abstraction is a Bicycle. It could defined as a seat, a frame, handlebars, and wheels where we ignore chains, pedals, brakes, etc..

1.1.5 Classification

Classification is used for defining one concept as a class of real-world objects characterised by some common properties. We often represent class instances with a dashed arrow pointing to the concept/classification. Figure 3 is an example of classification of a Month with January, February, and December as class instances.

We represent classification as a one-level tree having as its root the class, and as its leaves the elements of the class (see Figure 3); arcs of the tree are dashed. Each arc in the tree states that a leaf node is a member of (IS_MEMBER_OF) the class represented by the root.

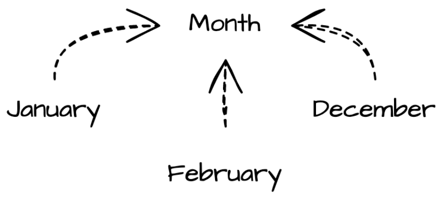


Figure 3: An Example of Classification

1.1.6 Aggregation

Aggregation defines one concept as a set of other concepts that represent its component parts. Observe that aggregation can be nested. We often represent component concepts with double or thick arrows pointing to the aggregation concept. Figure 4 shows a component concept (wheel, handlebar, and frame) for a Bicycle. Each arc in the tree states that a leaf class is a part of (IS_PART_OF) the class represented by the root. To distinguish aggregation from classification, the directed arcs are represented by thick (double) lines from component to aggregate objects.

In general, one can define an aggregation among *n* different classes. It is always possible to decompose an **n-ary** aggregation into (a larger set of) binary aggregations.

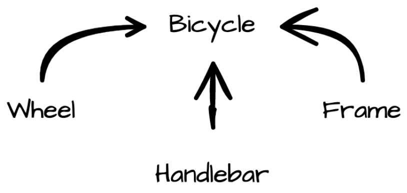


Figure 4: An Example of Aggregation

1.1.7 Generalisation

Generalisation defines one concept as being a more specific type of another concept (inheritance in object-oriented programming). All properties of the parent concept are inherited by all of the children. We often represent sub classes with a split arrow pointing to the parent concept. The example in Figure 5 generalizes a Person with sub classes such as Man and Woman. (Note: this is an outdated example of a Person).

Each generalization is represented by a one-level tree in which all nodes are classes, with the generic class as the root and the subset classes as leaves; each arc in the tree states that a leaf class is a (IS_A) subset of the root class. In a generalization, all the abstractions defined for the generic class are inherited by all the subset classes.

Coverage Properties There are multiple ways in which a concept can be a generalisation of other concepts.

- *Exclusive*: Elements of one sub-class cannot be part of another sub-class.
- *Overlapping*: Elements of the parent class can be members of multiple sub-classes.
- *Partial*: Some elements of the parent class do not belong to any sub-classes.
- *Total*: Every parent class concept is in at last one sub-class.

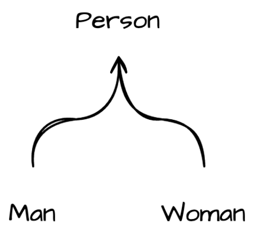


Figure 5: An Example of Generalisation

1.1.8 Composability

One key strength of these abstraction mechanisms is that they can be composed to model arbitrary complexity. Figure 6 creates a sub class of Man for a Person (with attributes of Name, Sex, Position) with an attribute of Draft_Status. Thus, Man has access to Name, Sex, and Position.

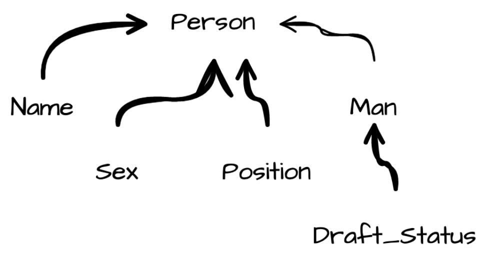


Figure 6: Composability

1.1.9 Binary Aggregations

A binary aggregation is a mapping between two classes. Note that we can draw a mapping as a bipartite graph.

Figure 7 shows a binary aggregation of Person and Building. Here we can see that a person can use a building and/or own a building. And likewise, a building can be used by a person and/or owned by a person. In this mapping we do not state whether a building can be used by multiple persons or if each person needs to own a building, etc..

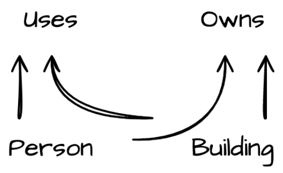


Figure 7: Binary Aggregation

1.1.10 Cardinality

Minimal Cardinality (min-card) Let us consider the aggregation A between classes C_1, and C_2 . The minimal cardinality, or min-card, of C_1, in A, denoted as min-card(C_1, A), is the minimum number of mappings in which each element of C_1 can participate. Similarly, the min-card of C_2 in A, denoted as min-card(C_2, A), is the minimum number of mappings in which each element of C_2 can participate.

If min-card(C_1,A) = 0, then we say that class C_1, has an **optional participation** in the aggregation, because some elements of class C_1 may not be mapped by aggregation A to elements of class C_2. If min-card(C_1, A) > 0, then we say that class C_1, has a **mandatory participation** in the aggregation, because each element of class C_1, must correspond with at least one element of class C_2.

Maximal Cardinality (max-card) Let us consider the aggregation A between classes C_1, and C_2 . The maximal cardinality, or max-card, of C_1 in A, denoted as max-card(C_1, A), is the maximum number of mappings in which each element of C_1 can participate. Similarly, the max-card of C_2 in A, denoted as max-card(C_2, A), is the maximum number of mappings in which each element of C_2 can participate.

If max-card(C_1, A) = 1 and max-card(C_2, A) = 1, then we say that the aggregation is **one-to-one**; if max-card(C_1, A) = n and max-card(C_2, A) = 1, then the aggregation C_1 to C_2 is **one-to-many**; if max-card(C_1, A) = 1 and max-card(C_2, A) = n, then the aggregation C_1 to C_2 is **many-to-one**; finally, if max-card(C_1, A) = m and max-card(C_2, A) = n (where m and n stand for any number greater than 1), then the aggregation is **many-to-many**.

Multiplicities In general, we observe four types of multiplicities, or pairs of cardinalities in a binary aggregation. There is one-to-one, one-to-many, many-to-one, and many-to-many.

Minimal and Maximal Cardinality Let A be a binary aggregation of classes C_1 and C_2, with min-card(C_1,A) = m_1, and max-card(C_1, A) = M_1; then we say that the cardinality of C_1 in A is the pair (m_1, M_1): card(C_1, A) = (m_1, M_1). Similarly, if min-card(C_2, A) = m_2 and max-card(C_2, A) = M_2, then card(C_2, A) = (m_2, M_2).

1.2 The Entity-Relationship Model

Lesson Scope

This lesson covers §4.0–4.2 of Garcia-Molina et al. (2009) [2] and §2.3–2.4.1 of Batini et al. (1992) [1], which overlap significantly. It will introduce the basic elements of the Entity-Relationship Diagram (ERD), a ubiquitous model for communicating conceptual schemata.

Lesson Objectives

- When asked, you can differentiate the key elements of a conceptual or logical data model, including the role of relational theory.
- When you create a conceptual design, you can communicate it the key elements (entities, relationships, attributes, and multiplicities) using an accurate entity-relationship diagram (ERD).
- Given an entity-relationship diagram (ERD), you can accurately identify the key elements (entities, relationships, attributes, and multiplicities).

Delivery Method

This lesson will primarily follow the corresponding slide deck, which takes content on the "qualities" of the ERD model from Batini et al. [1] and most of the other content from Garcia-Molina et al. [2]. If there is extra time, we will discuss the differences between the basic elements of the ERD model presented in each of the two textbooks.

Preparation

You are encouraged to read the sections of the textbook in advance, but this is not required to follow the lecture.

1.2.1 Entity-Relationship Diagrams (ERD's)

An ERD is a conceptual model that only describes broadly the entities in a database and their relationships. It is *Expressive*, *Simple*, *Minimal*, and *Formal*.

1.2.2 Entity Sets

Entities represent classes of real-world objects. Entities are graphically represented by means of rectangles. In Figure 8 we see an entity set of Student.

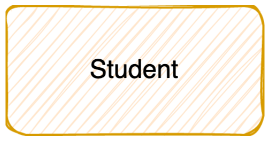


Figure 8: An Example of an Entity Set

1.2.3 Attributes

Attributes represent elementary properties of entities or relationships. All the extensional information is carried by attributes. They are denoted by an circle with the name of the attribute inscribed therein and a line connecting it to the relevant entity set. Figure 9 shows Student having attributes V_number and Name.

Let A be an attribute of entity E; if min-card(A, E) = 0, then the attribute is optional, and it can be unspecified (null) for some entity instances. If instead min-card(A, E) = 1, then the attribute is mandatory, and at least one attribute value must be specified for all entity instances.

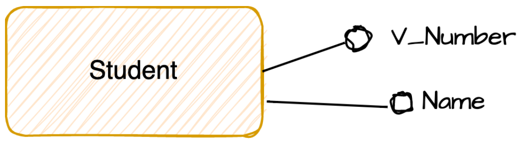


Figure 9: An Example of Attributes On An Entity Set

1.2.4 Relationships

Relationships represent aggregations of two or more entities. N-ary relationships connect more than two entities. They are denoted by a diamond with the name of the relationship inscribed therein and a line connecting it to the relevant entity sets. We see a relationship of Enrolled_In between Student and Class in Figure 10.

Attributes represent elementary properties of entities or relationships. All the extensional information is carried by attributes. Thus, it is possible for a relationship to have an attribute if that attribute describes the connection between the entity sets rather than the entity set itself. In Figure 11 we see that the relationship Enrolled_In has an attribute of grade. We do this because a Student doesn't have a single grade because grade is dependant on the enrollment in a class.

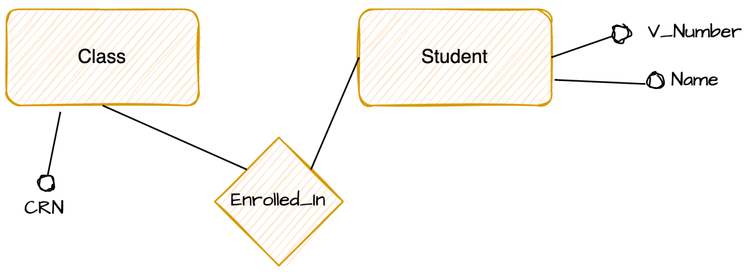


Figure 10: A Relationship Between Student and Class

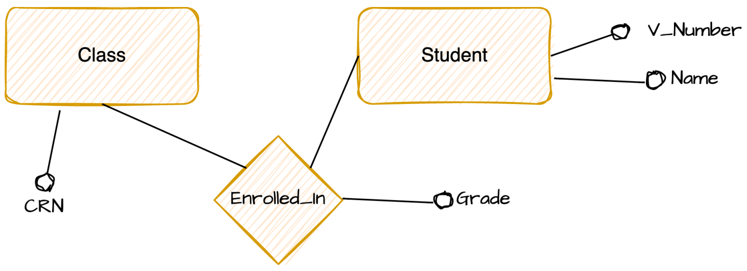


Figure 11: A Relationship Enrolled_In With An Attribute Grade

1.2.5 Relationships Multiplicity

A relationship has a multiplicity that describes how often entity sets participate with each other in the relationships.

- **many-many** is denoted with a simple connector. (As observed above in the examples).
- **one-many** is denoted by a directional arrow from one side of the relationship.
- **one-one** is denoted by a direction arrow from both sides of the relationship.

1.2.6 Relationships Roles

Relationship roles is how an entity set participates in a relationship. It can be described with a text label called role. This is especially helpful when an entity set participates more than once in a relationship (also known as a recursive relationships). Figure 12 is an example of a relationship Tutor with Student. With distinction of tutor and tutee to clarify the self loop.

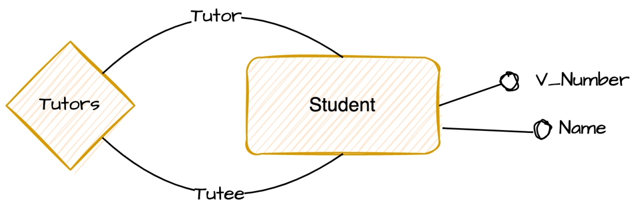


Figure 12: A Recursive Relationship Tutor with Student

1.3 Identifiers, Weak Entities & Sub-classes

Lesson Scope

This lesson covers §2.4.2–2.6 of Batini et al. (1992) [1], which also overlaps §4.4 and §4.1.11 of Garcia-Molina et al. (2009) [2]. It will introduce more complex aspects of entity-relationship diagrams (ERD's), such as generalisation hierarchies and entity sets who are partly identified by other entity sets ("weak entities").

Lesson Objectives

1. Given a generalisation abstraction, you can model it accurately in an ERD using notation for subsets and specialisations.
2. Given an ERD with hierarchical relationships, you can accurately describe the generalisation abstraction that it models.
3. Given an entity set, you can accurately model appropriate identifiers in an ERD, including those that depend on other entity sets.
4. Given an ERD, you can accurately describe all identifiers, including those of weak entity sets.

Delivery Method

This lesson will walk-through the accompany slide deck, which in turn follows the remainder of Chapter 2 in Batini et al. [1].

Preparation

You are encouraged to read the sections of the textbook in advance, but this is not required to follow the lecture.

1.3.1 Basic ERD Notation

Relationships are characterized in terms of minimal and maximal cardinalities. We synthesize minimal and maximal cardinality as a pair of values.

Figure 13 has card(PERSON, LIVES_IN) = (1, 1) and card(CITY, LIVES_IN) = (0, n). Thus, each Person is related to one and only one city. A City can have no inhabitants. Each pair is represented on the schema, close to the connection between the entity and the relationship.

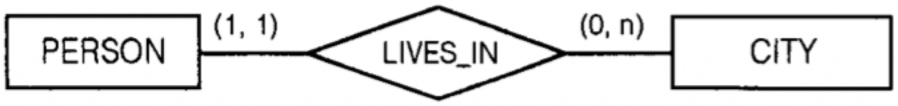


Figure 13: Relationship LIVES_IN

The ternary "meets" relationship, shown in Figure 14, exists between a Course, a Classroom, and a Day. Every Course must meet at least once but not more than 3. A Course does not need to meet every Day. A Classroom can be used between 0 or 40 times.

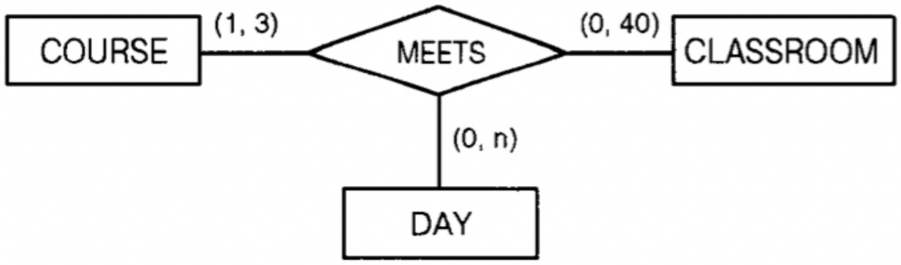


Figure 14: Relationship MEETS

The relationship MANAGES in Figure 15 (a one-to-many relationship, since each manager manages many employees, and each employee has just one manager; participation in the relationship is optional).

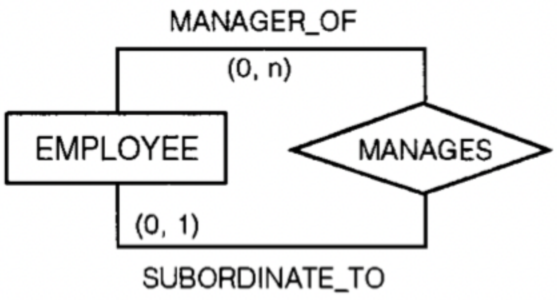


Figure 15: Relationship MANAGES

We observe in Figure 16 that a Person is represented as a name, Social_Security_Number, Profession, and a list of Degrees. A City is represented as a Name, Elevation, and Number_Of_Inhabitants.

Each Person is born in at most one City, but not necessarily any. There is no restriction on how many people are born in each City. Each birth is associated with a Birth_Date.

Persons are also related to Cities though the "Lives_In" relationship which is characterised by a Moving_Date. There are no restrictions at all on the cardinalities (a.k.a multiplicities) or "Lives_In".

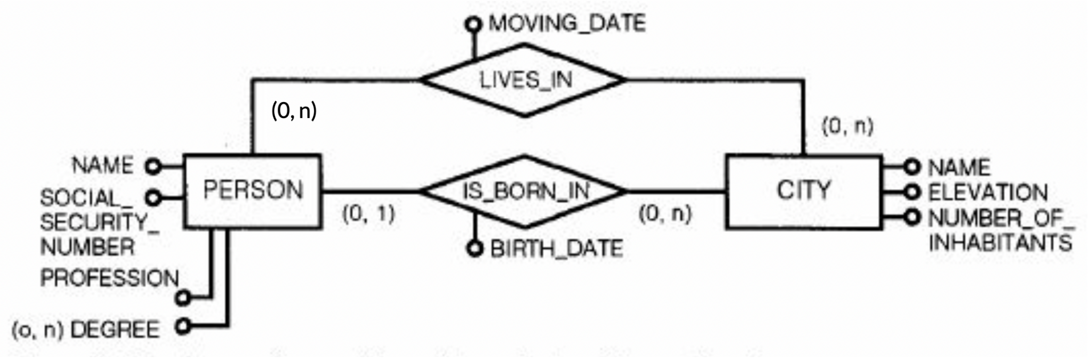


Figure 16: An ER Schema With Entities, Relationships, And Attributes

We observe that the two entity sets represent classification abstractions; The attributes and relationships represent aggregation abstractions.

1.3.2 Generalisation Hierarchies

An entity set E is a generalisation of a set of other entity sets $E_1, E_2, ..., E_n$ if and only if... $E_1 \cup E_2 \cup ... \cup E_n \subseteq E$. That is to say, every entity that appears in any of the specialisation entity sets (i.e., sub-classes) is also an entity of the generalisation entity set (i.e., parent class), but not necessarily vice versa. The opposite of generalization is referred to as specialization.

As with the generalisation abstraction, we can represent this in ERD with a multi-pronged arrow in the direction of the parent/generalisation. Multiple inheritance is permitted.

The generalization abstraction is captured by generalization hierarchies or subsets. It is usually applied only to entities, although some extensions of the ER model apply the generalization abstraction also to relationships or attributes.

1.3.3 Inheritance in Generalisation Hierarchies

Recall the inheritance property and its unidirectionality. For example, in Figure 17, every Sales Employee has all the attributes of Person, but none of those unique to the Male or Female entity set.

Note: You want to avoid having redundancy in your ERD's. For example if both the parent class and the sub-class(es) have Name as an attribute, it would make sense to remove the Name attribute from the sub-class(es). As we know that it is inherited already from the general entity set. It is also advised to have specific attributes on their respective sub-class(es) instead of an optional attribute on the parent class.

1.3.4 Coverage of Property of Generalisations

Recall that two "coverage" properties characterise the generalisation abstraction:

- Total vs Partial (t vs p): whether or not every general entity must also be at least one specialised entity.
- Exclusive vs Overlapping (e vs o): whether or not every general entity can be more than one specialised entity.

To indicate this in an ERD, we label an arrow with an ordered pair (t, e), (t, o), (p, e), or (p, o). Total and exclusive is assumed by default if no label is present.

- Total, exclusive (t,e): Exactly 1
- Partial, overlapping (p, o): No constraints
- Partial, exclusive (p, e): At most 1
- Total, overlapping: At least 1

Here in Figure 17 every Person is exclusively either Male or Female. Some Persons are exclusively either a Manager, Secretary, or Employee. Some Managers may be a Technical Manager, Administrative Manager, or both. Employees can have any subset of the following roles: Programmer, Sales Employee, or Advertising Employee.

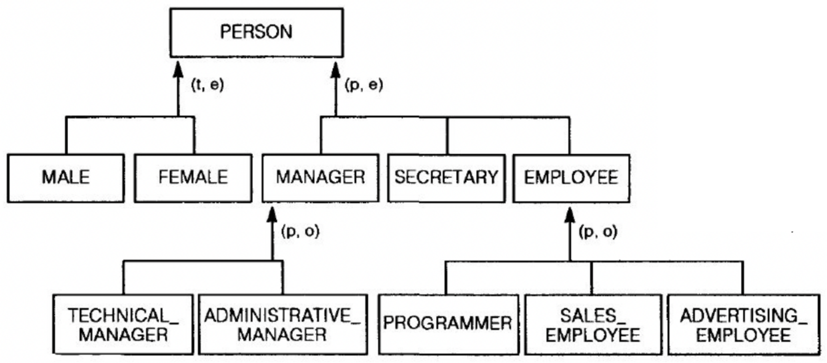


Figure 17: Generalization Hierarchy for the Entity PERSON

1.3.5 Special Generalisation Case: Subsets

There is a special case when there is only one specialisation. We call this a *subset*. There is no multi-pronged arrow, because there is only one child. There is no label, because it is by default (p,e). This arises specifically when there is a subset of an entity set to which additional attributes or an additional relationship applies.

1.3.6 Composite Attributes

We apply the aggregation abstraction to attributes when we want to treat them as a group. We indicate this with an oval around the composite.

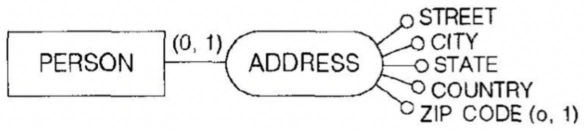


Figure 18: Composite, Multivalued Attribute

In the example presented in Figure 18 we create a *composite* attribute for Address, which has component fields: Street, City, State, Country, and Zip Code. Observe that this way we can specify a cardinality for the composite group (either zero or one Address), rather than mimicking that with a cardinality on each component group.

1.3.7 Identifiers

A set of attributes A is an identifier of entity set E if and only if: There cannot be two instances of E that have the same value for all attributes in A. And A is minimal (i.e., no subset of A meets the first condition). We denote an identifier with a filled-in circle. We should assign an identifier to every entity set. Figure 19 is an example of assigning an attribute of Social_Security_Number to a Person as an identifier.

Since identification is a property of entities, it is one of the properties inherited in generalizations or subsets: the identifier of the generic entity is also an identifier for the subset entities.

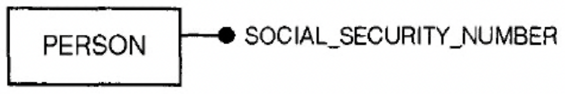


Figure 19: Simple, Internal Identifier

Here, it is not possible for two Persons to have the same Social_Security_Number (SSN). Thus {SSN} is a “simple” (i.e., single-attribute) identifier.

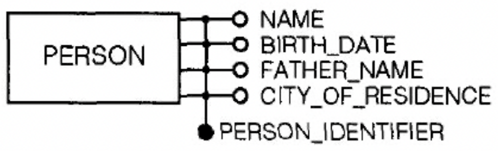


Figure 20: Composite, Internal Identifier

Here in Figure 20, any two Persons could have the same Name, Birth_Date, and City_Of_Residence, but they cannot also have the same Father_Name. This identifier is composite, because the identifier has many attributes.

1.3.8 Weak Entity Sets

A *strong entity* set E, is one in which the identifiers are a subset of the attributes E.

A *weak entity* set E, instead, has a apart of its identifier attributes that come from some other entity E′. A weak entity set borrows the entire key of some entity set to which it is related. Thus, we build a composite attribute that connects to the relationship line.

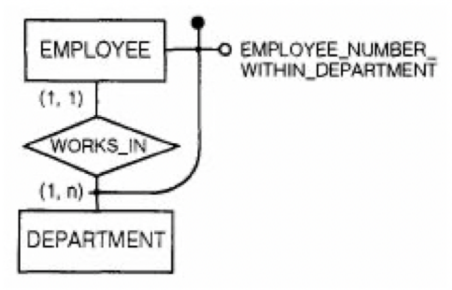


Figure 21: Composite, External, Mixed Identifier

In Figure 21 an Employee has Employee_Number_Within_Department and only works for one Department. The Employee identifier is thus the composite attribute (Department and Employee_Number_Within_Department).

Note: Here we are using Department, but it is incorrect because the diagram is incorrect. The composite attribute should be (Department Identifier Attribute(s) and Employee_Number_Within_Department).

1.3.9 Abstraction Mechanisms in the ER Model

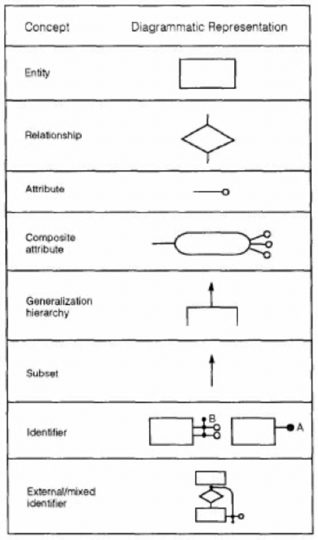
It is instructive to show how the three abstractions are captured by the concepts of the ER model. We summarizes the graphic symbols used in the ERD in Figure 22.

Classification Abstraction The three basic concepts of the ER model are developed as applications of the classification abstraction:

1. An *entity* is a class of real-world objects with common properties.
2. A *relationship* is a class of atomic (elementary) facts that relate two or more entities.
3. An *attribute* is a class of values representing atomic properties of entities or relationships.

Aggregation Abstraction Three types of aggregations characterize the ER model:

1. An *entity* is an aggregation of attributes.
2. A *relationship* is an aggregation of entities and attributes.
3. A *composite attribute* is an aggregation of attributes.



Concept	Diagrammatic Representation
Entity	
Relationship	
Attribute	
Composite attribute	
Generalization hierarchy	
Subset	
Identifier	
External/mixed identifier	

Figure 22: Graphic Symbols Used in the ERD

4. An *(externalised) identifier* is an aggregation of attributes that are unique to instances of an entity set.

Generalisation Abstraction A generalisation hierarchy (or subset) is a generalisation of one entity set from a collection of other entity sets.

1.3.10 Reading ERD's

Consider the schema in Figure 23 which represents properties of persons within universities. How do we interpret the schema? Where do we even start? A logical approach would be to start with generalisation abstractions and associated relationships. They usually represent meaningful clusters of information. Next we can expand outwards from this point. For example, we can take note that we have Person that is split into Student and Professor. And a Student can be a Graduate Student. We can also see that Students are Residents_Of Cities and are Enrolled in Courses.

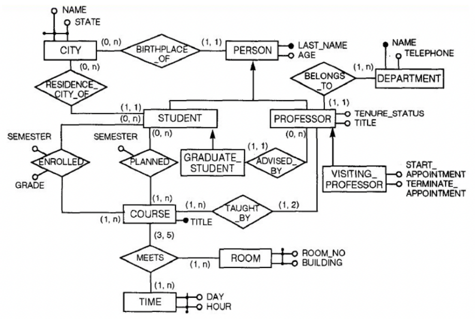


Figure 23: University Database

The reading of the schema is completed by examining the properties of individual entities and relationships. Identifiers in the schema are quite simple. They are all internal; City, Room, and Time have composite identifiers, while all other identifiers are simple.

1.3.11 Recap and Summary

This chapter has introduced data modeling concepts, including data abstractions, general features of conceptual models, and the Entity-Relationship model. In particular, we have introduced the elements of the ER model: entities, relationships, attributes, composite attributes, generalization hierarchies, subsets, and identifiers. This chapter has emphasized the understanding of a given, existing ER schema; with this background, we can turn to the problem of modeling reality in a conceptual schema.

We conclude with one more example, shown in Figure 24.

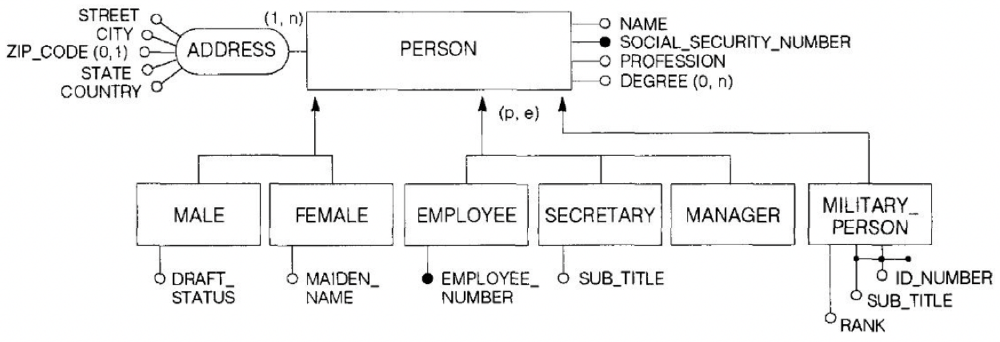


Figure 24: Generalization Hierarchies, Composite Attributes, And Identifier In the ER Model

We see that a Person is identified by a Social_Security_Number (SSN) and also represented by a Name, Profession, list of Degrees, and at least one Address. Every Person is either Male or Female; the latter has a Draft_Status and the former, a Maiden_Name. Some Persons are Military_Person who can be identified by an SSN or by the combination of Id_Number and Sub_Title. They also have a Rank. Everyone is exclusively either a Manager, a Secretary (who has a Sub_Title), an Employee (who can alternatively be identified by their Employee_Number) or none of the above. Each Address is a composition of a Street, City, State, Country, and an optional Zip Code.

1.4 Design Methods

Lesson Scope

This lesson covers §3–3.2 of Batini et al. (1992) [1]. It will answer the question, “Now that we know what to draw and read ER diagrams, how do we create them from a set of requirements?” We will investigate three design methods: top-down, bottom-up, and a mixed strategy. Moreover, we will look at a simple case study and solve it using all three methods.

Lesson Objectives

1. Given a set of requirements in natural language, you can apply a rigorous and flexible design method to convert it into a conceptual schema that captures those requirements, expressed as an entity-relationship diagram.

Delivery Method

We will follow the corresponding slide deck. Several slides are in-class exercises that will follow a think-pair-share pedagogical method where you will be given about three minutes to solve it with your neighbours in the classroom before I ask for volunteers to share their approach. Because the lesson is design focused, the sharing aspect is meant to demonstrate the diversity of possible approaches. You are welcome to work on such problems alone and you are not required to share your ideas, though that may limit the diversity of what is shown. Afterwards, I will reveal the textbook solution, which is one possible (and obviously very good) solution. Such activities will not take up more than fifteen minutes of the scheduled lecture time.

It is anticipated that this lesson may span more up to two lecture blocks.

Preparation

You are encouraged to read the sections of the textbook in advance, but this is not required to follow the lecture.

1.4.1 Design Methods

A good design process has at least two properties...

- *Rigorous*: It suggests a process for making all important decisions in the design process, based on a formal approach, perhaps even corresponding to an algorithm.
- *Flexible*: It should be applicable to a wide variety of situations and environments so as to be adaptable to organisational constraints and a designer's style.

We will look at four design methods...

- *Top-Down*: It begins with an abstract concept and continually refines it into concrete ones.
- *Bottom-Up*: It begins with all the concrete concepts and continually combines them to capture the entire abstract concept.
- *Mixed*: Partitions the model into a skeleton schema and then refines the individual components into concrete concepts before linking them back together again.
- *Inside-Out*: Special case of top-down where we fix the most important concepts and then expand outwards from them "moving as an oil stain does".

1.4.2 Design by Transformation

All of these depend on defining a set of primitive operations that enable us to transition from one schema to a new schema. The overall process starts from a schema, then we apply transformations, and then we repeat until a desired schema is obtained. So all we need is a set of valid transformations and then we have a rigorous methodology. If those allow us to arrive at any possible schema, it is also a flexible methodology. These are the elementary building blocks for the development of a design methodology.

Each transformation has three characteristics...

- A starting schema (to which we apply the transformation) and a resulting schema (the product of applying the transformation).
- A mapping between concepts in the starting schema and the resulting schema.
- Inheritance of all logical connections that existed before.

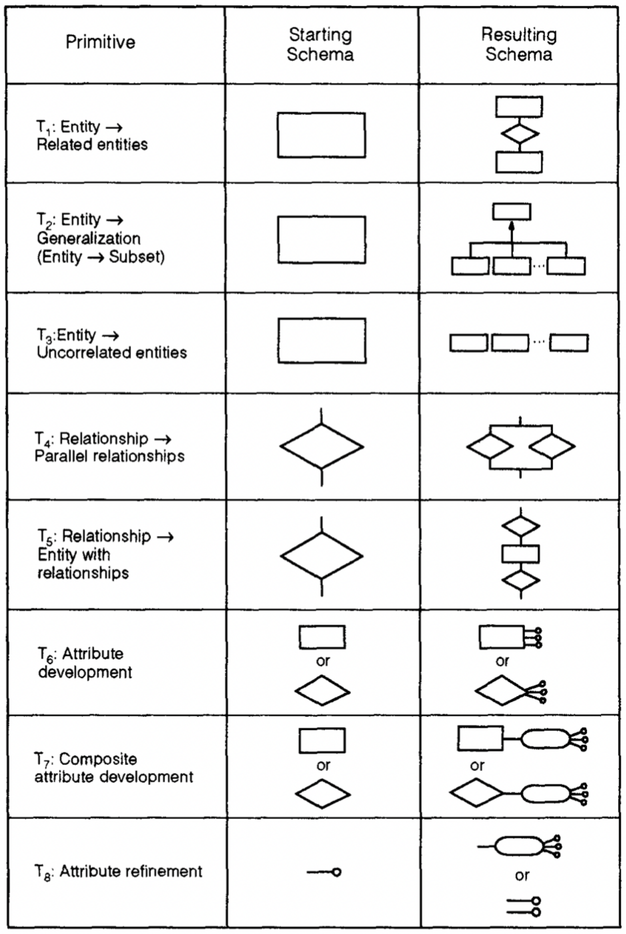
Primitives are classified in two groups, top-down and bottom-up. Top-down primitives correspond to pure refinements; that is, refinements that apply to a single concept (the starting schema) and produce a more detailed description of that concept (the resulting schema). By contrast, bottom-up primitives introduce new concepts and properties that do not appear in previous versions of the schema.

1.4.3 Top-Down Design Primitives

The Top-Down approach starts with an abstract statement of the application domain. And then we refine the model by applying Top-Down primitives.

We can think of it as: Application Domain $\rightarrow$ First Refinement Plane $\rightarrow$ Generic Refinement Plane $\rightarrow$ Final Refinement Plane.

Figure 25 shows an example of a transformation for each of the 8 primitives in the Top-Down approach.



Primitive	Starting Schema	Resulting Schema
T_1: Entity $\rightarrow$ Related entities		
T_2: Entity $\rightarrow$ Generalization (Entity $\rightarrow$ Subset)		
T_3:Entity $\rightarrow$ Uncorrelated entities		
T_4: Relationship $\rightarrow$ Parallel relationships		
T_5: Relationship $\rightarrow$ Entity with relationships		
T_6: Attribute development	or	or
T_7: Composite attribute development	or	or
T_8: Attribute refinement		or

Figure 25: Classification of Top-Down Primitives

1.4.4 Top-Down Design Primitives - Explanation

Each primitive performs a specific type of refinement on the schema.

- Primitive T_1: Refines an entity set into a relationship and at least two entity sets.
- Primitive T_2: Refines an entity set into a generalisation and at least one specialisation.

- Primitive T_3: Splits an entity set into two or more independent entity sets (i.e., with no relationship or hierarchy).
- Primitive T_4: Refines a relationship into two or more relationships between the same entity sets.
- Primitive T_5: Refines a relationship into a path with two or more relationships.
- Primitive T_6: Refines an entity set (or relationship) by adding attributes to it.
- Primitive T_7: Refines an entity set (or relationship) by adding a composite attribute to it.
- Primitive T_8: Refines an attribute by splitting it either into a set of attributes or a composite attribute.

1.4.5 Bottom-Up Design Primitives

In bottom-up design, we first compile a collection of disconnected, elementary concepts. And then until we reach the final schema: aggregate elementary concepts using a "bottom-up" primitive.

We can think of it as: Application Domain $\rightarrow$ Production of Elementary Concepts $\rightarrow$ Collection of Elementary Concepts $\rightarrow$ Aggregation of Elementary Concepts $\rightarrow$ Final Schema.

Figure 26 shows an example of a transformation for each of the 5 primitives in the Bottom-Up approach.

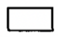
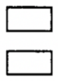
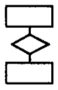
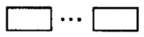
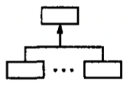
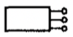
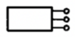
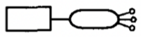
Primitive	Starting Schema	Resulting Schema
B_1: Entity generation		
B_2: Relationship generation		
B_3: Generalization generation (subset generation)		
B_4: Attribute aggregation		
B_5: Composite attribute aggregation		

Figure 26: Classification of Bottom-Up Primitives

1.4.6 Bottom-Up Design Primitives - Explanation

Bottom-up primitives introduce new concepts and properties that did not appear in previous versions of the schema, or they modify some existing concepts. Bottom-up primitives are used in the design of a schema whenever we discover features of the application domain that were not captured at any level of abstraction by the previous version of the schema. Bottom-up primitives are also applied when different schemas are merged into a more comprehensive, global schema.

- Primitive B_1: Generates a new entity. It is used when a designer discovers a new concept with specific properties that was not in the previous schema.
- Primitive B_2: Creates a new relationship between previously defined entity sets.
- Primitive B_3: Creates a new entity set that is elected as a generalisation of previously defined entity sets. Creates a new entity set that is elected as a generalisation of previously defined entity sets. **Note that B_3 requires checking if attributes should migrate to the generalisation concept.**
- Primitive B_4: Generates a new attribute and and connects it to a previously defined relationship or entity set.
- Primitive B_5: Creates a composite attribute and connects it to a previously defined entity set or relationship.

1.4.7 Mixed Strategy Design

In a mixed strategy design, we: partition the original domain into disjoint components connected via a skeleton schema (i.e., divide and conquer).

For each partition: apply the top-down methodology. And then use the bottom-up methodology to re-link the partitions.

We can think of it as: Application Domain $\rightarrow$ Application Domain 1 $\rightarrow$ Schema 1 $\rightarrow$ Integrated Schema. Application Domain $\rightarrow$ Application Domain 2 $\rightarrow$ Schema 2 $\rightarrow$ Integrated Schema. And Application Domain $\rightarrow$ Skeleton Schema $\rightarrow$ Integrated Schema.

1.4.8 Comparison of Strategies

Starting from given requirements, do the four strategies always lead to the same final schema? Of course the answer is negative, since each strategy follows a specific design philosophy. It is important to perform periodically a quality check on the schema.

We discuss the advantages and disadvantages to each of the 4 methods for designing ERD models in Table 4.

Each design session should employ the most convenient strategy with respect to the specific environment. For example, a top-down strategy may be most convenient with highly structured organizations, where top management has a complete view of the application domain at a high level of abstraction. On the other hand, a bottom-up strategy may be convenient with informal and loosely structured organizations, where modeling detailed information and then aggregating it is easier than building an initial abstraction of the entire database and then refining it.

Strategy	Description	Advantages	Disadvantages
Top-Down	Concepts are progressively refined.	No undesired side effects.	Requires a capable designer with high abstraction ability at the very beginning.
Bottom-Up	Concepts are built from elementary components.	Ease of local design decisions.	Need restructuring after applying each bottom-up primitive.
Inside-Out	Concepts are built with an oil-stain approach.	Ease of discovering new concepts close to previous ones. No burden on the initial designer.	A global view of the application domain is built only at the end.
Mixed	Top-Down partitioning of requirements; Bottom-Up integration using a skeleton schema.	Divide and conquer approach.	Requires critical decisions about the skeleton schema at the beginning of the design process.

Table 4: Comparing of the Strategies for Schema Design

We conclude that the top-down strategy should be used whenever it is feasible. Otherwise we should adopt a mix of strategies-those that appear the most natural in the specific context of the application. A pure top-down strategy can be applied to most real-life problems.

1.5 Design Qualities

Lesson Scope

This lesson covers §6–6.5 of Batini et al. (1992) [1]. It will answer the question, "How do we assess or improve the quality of a conceptual design?" We will investigate eight qualities of a good design: completeness, correctness, minimality, expressiveness, readability, self-explanation, extensibility, and normality. Moreover, we will wrap up the unit on conceptual design with a preview of why normality is important and how the normalisation techniques of logical design in the relational data model improve data quality.

Lesson Objectives

1. Given a conceptual design expressed as an Entity-Relationship Diagram (ERD), you can apply information-preserving transformations to improve the quality of the design.

Delivery Method

We will follow the corresponding slide deck. Several slides are group exercises that will solve as a whole class.

Preparation

You are encouraged to read the sections of the textbook in advance, but this is not required to follow the lecture.

1.5.1 Quality Design

What aspects would we expect of a *good* conceptual design?

- **Completeness**: It represents all features of the application domain. In principle, this can be checked by comparing all requirements of the application domain and checking that they are captured somewhere in the conceptual schema.
- **Correctness**: It properly uses the concepts of the ER model. Syntactically correct when the right notions are used (e.g., no generalisation of relationships); It is semantically correct if the notions are used correctly. The typical errors that impact the correctness are: using an attribute instead of an entity set (or vice versa), forgetting a generalisation or implied inheritance, a relationship with too few/many entity sets, and forgetting an identifier or min/max specification.
- **Minimality**: Every aspect of the schema appears in the requirements and is represented in the schema only once. It is minimal if no element can be deleted without violating the requirements or losing some information. (No complexity, try to keep it simple).
- **Expressiveness**: It represents requirements in a natural way and can be easily understood without explanation. (Subjective).
- **Readability**: It respects certain aesthetic criteria that make the diagram graceful, say by: drawing the diagram on a grid with horizontal/vertical lines, using symmetry, few line crossings, and few line bends, and parents should be drawn above children in the diagram. (It respects the language direction).
- **Self-Explanation**: A large number of annotations can be expressed with the schema itself, without, say, text annotations.
- **Extensibility**: It is easily changed to adapt to changing requirements, e.g., because it is decomposed into pieces (modules or views) so that changes are localised.
- **Normality**: It would, if represented as a relational database, fit into standard normal forms like Boyce-Codd, 3NF, or 4NF.

1.5.2 Achieving Minimality

There are many ways to achieve minimality; cycles, derived attributes, implicit subsets, dangling sub-entities, and dangling entities.

Cycles Redundancy exists when one relationship R_1 between two entities has the same information content as a path of relationships R_2, R_3, ..., R_k connecting exactly the same pairs of entity instances as R_1.

Derived Attributes Redundancy may be due to the existence of an algorithm for computing the values of derived data from the other data; hence derived data can be omitted from a minimal ER schema (i.e., it can be calculated, usually attributes such as “total”).

Implicit Subsets Redundancy exists when some of the subsets might be derived from other subsets present in the schema.

Dangling Sub-Entities It may happen that the designer creates a generalization in the process of assigning different properties to entities in the hierarchy. If, at the end of the design process, the sub-entities are not distinguished by any specific property, they can be reduced to the super-entity.

Dangling Entities We regard an entity E as dangling if it has few (possibly one) attributes A, and one connection to another entity (the principal entity).

1.5.3 Schema Transformations

A schema transformation produces a new schema S_2 from an input schema S_1. We classify them based on the *information content*:

- **Information-Preserving**: The information content of the schema is not changed by the transformation. (We use information-preserving ones to improve structure, readability, and/or quality).
- **Information-Changing**:
  - *Augmenting*: The information content of the resulting schema is greater than that of the input schema. (We usually used augmenting transformations in design).
  - *Reducing*: The information content of the resulting schema is less than that of the input schema. (We use reducing transformations to eliminate superfluous information).
  - *Incomparable*: Otherwise.

1.5.4 Achieving Normalisation

One important quality is *normalisation* (one of the strengths of the relational data model).

A few anomalies that can happen are update anomalies, insertion anomalies, and deletion anomalies.

1.6 Midterm 01

1.6.1 Scope

This exam covers the conceptual design unit of the course, including Chapter 4 of the Garcia-Molina et al. textbook [2] and Chapters 1-3 and 6 of the Batini et al. textbook [1]. This overlaps all class lectures strictly earlier than Thursday, 29 September. The exam assesses your knowledge of conceptual database design, particularly that:

- given an Entity-Relationship Diagram (ERD), you can accurately articulate the meaning of components thereof, including attributes, entity sets, relationships, multiplicities, abstraction mechanisms, and identifiers.
- given an Entity-Relationship Diagram (ERD) and a set of natural language requirements, you can indicate whether the ERD is complete and minimal with respect to those requirements.
- given two schemata, you can indicate whether one improves a design quality of the other by means of an information-preserving transformation.

1.6.2 Format

This *closed-book, multiple choice exam* is written in-person with responses recorded on a standard five-option UVic bubble sheet. You may bring to the exam one single-sided A4 or US Letter page of hand-written or typed notes. The practice exam below is highly predictive of the style of questions that you can expect. You have up to forty-five minutes to write the exam, after which we will take a ten minute break and then resume with a shortened lecture. The exam will not be live-streamed over Zoom nor recorded, but the lecture thereafter will be.

For *online-first students (A03)*, please note that there have been sufficiently many course withdrawals and CAL registrations that there is space for everyone to write the exam in the lecture theatre, even without considering that some students may choose to only write the assignment. Please feel free to join us if you want.

2 Design Theory for Relational Databases

Overview

At the level of logical design, we consider how abstract concepts should actually be structured, constrained, and manipulated as data.

A ubiquitous logical data model, the one on which this course focuses, is the relational data model. A relational database structures data as a series of well-chosen two-dimensional tables (called relations), similar to a set of strongly-typed spreadsheets.

As a closed model, it permits a small set of operations that take one or more relations and produce a relation: in this way, we can compose and nest operators to build up arbitrary complexity. The relational data model also comes with a language, relational algebra, in which we can express constraints on data using the model's operators. This gives us a robust theoretical toolkit to reason about what sort of "anomalies" (i.e., data quality errors) could arise in a relational database.

We will dive deep into the theoretical foundations of the relational data model so that you can design a normalised relational database that will flatly reject many classic sources of data entry errors.

Module-Level Intended Learning Outcomes

Upon completion of this short introductory module, you can:

- given a conceptual model or a requirements specification, identify and specify in relational algebra all relevant constraints, including functional dependencies and referential integrity constraints.
- given a set of relations and a set of functional or multi-value dependencies, minimally decompose the data model into Third, Boyce-Codd, and Fourth normal form.
- given a decomposition of a relation and a set of functional dependencies (FD's), identify which FD's project onto each sub-relation and which are lost.

References

Garcia-Molina, Ullman and Widom (2009) [2]. Database Systems: The Complete Book. Pearson Prentice Hall. Chapters 2–3.

2.1 Relations and Functional Dependencies

Lesson Scope

The relational data model is a logical model that has become ubiquitous in database systems since it was introduced by Edgar F. Codd in 1970. It structures data as a collection of two-dimensional tables, called relations, and introduces a powerful algebraic language for expressing constraints. A specific type of constraint, called a functional dependency, is central to the data model and leads directly to the choice of which relations should be used in our logical schema. This lesson covers §2.2 and §3.1 of Garcia-Molina et al. (2007) and will review the basic definitions of relations and functional dependencies.

Lesson Objectives

1. When discussing concepts in this course, you can use precise terminology for basic concepts in the relational data model, such as "schemata", "tuples", "relations", "attributes", and "instances".
2. Given a set of attributes, you can identify a realistic set of functional dependencies.
3. When asked, you can formally define a functional dependency.

Delivery Method

We will follow the corresponding slide deck. A couple slides are in-class exercises that will follow a think-pair-share pedagogical method where you will be given about three minutes to solve it with your neighbours in the classroom before I ask for volunteers to share their approach. Because some aspects of the lesson are design focused, the sharing aspect is meant to demonstrate the diversity of possible approaches. You are welcome to work on such problems alone and you are not required to share your ideas, though that may limit the diversity of what is shown. Afterwards, I will reveal my prepared solution, which is one possible (and obviously very good) solution. Such activities will not take up more than ten minutes of the scheduled lecture time.

Preparation

You are encouraged to read through the sections of the textbook in advance, but the lesson will not assume that you have done so.

2.1.1 Relational Data Model

"A data model is a notation for describing data or information," generally composed of three parts:

1. **A fixed type of structure**. The relational data model structures data as a set of two-dimensional tables called relations.

2. **A set of permitted operators**. We use a combination of set theory and table-oriented operations that filter out information or merge relations together.

3. **A language for expressing constraints**. The relational data model comes with an algebraic language, called relational algebra, that enables us to express very complex constraints in terms of relational operators.

Table 5 is an example of an instance of a relation. Here we can use a selection operator. We could use a predicate of year=1939 to remove the last two rows. For example, $\sigma_{year \neq 1939}(Movies) = \emptyset$ to express that all movies must be from 1939.

We can think of a relation as a definition of a two-dimensional table. A relation instance is a two-dimensional table with data (as seen in Table 5).

title	year	length	genre
Gone With the Wind	1939	231	drama
Star Wars	1977	124	sciFi
Wayne's World	1992	95	comedy

Table 5: An Instance of A Relation - The Relation Movies

2.1.2 Nomenclature of Relations: Attributes

An attribute is the name of a column in the two-dimensional table. A relation has a set of attributes. Thus, they are unordered. However, it is customary to use a consistent ordering for readability.

2.1.3 Nomenclature of Relations: Schema

The schema of a relation is the combination of its name and its attribute set. The schema of a database is a set of relation schemata.

For example, Movies(title, year, length, genre) from Table 5 where the order of title, year, length, and genre does not matter.

2.1.4 Nomenclature of Relations: Tuples

A tuple is a row of the two-dimensional table. A relation instance has a set of tuples. Note: It is not possible for the same tuple to appear twice, nor does the order of tuples have any meaning. We often refer to arbitrary tuples by a variable. For example, t, u, and v or t_i and t_j.

2.1.5 Nomenclature of Relations: Domains

Each attribute of a relation has a fixed domain (data type). The domain must be an elementary type (relations cannot be nested and attributes cannot be objects). The data types must be respected to avoid violating the schema.

Thus, we have Movies(title:string, year:integer, length:integer, genre:string) for Table 5.

2.1.6 Nomenclature of Relations: Keys

Given a relation R that has a set of attributes A...

- A key K is a subset of A ($K \subseteq A$) where no two tuples can have the same values for all attributes $A_i \in K$.

We identify a key of the relation by underlining the attributes that are part of the key. In Table 5 we would then have Movies($\underline{\text{title}}$, $\underline{\text{year}}$, length, genre).

2.1.7 Functional Dependency

Consider a schema $R(C_1, C_2, ..., C_k)$ and let $C = \{C_1, C_2, ..., C_k\}$ be a set of attributes. Let $A = \{A_1, A_2, ..., A_n\} \subseteq C$ and $B = \{B_1, B_2, ..., B_m\} \subseteq C$ be subsets of R's attributes. We say that A functionally determines B, denoted A $\rightarrow$ B, if and only if...

For every two possible tuples of R, if they have the same values for all attributes in A, then they must have the same values for all attributes in B.

Table 6 shows a visual of the effect of a functional dependency on two tuples.

Functional dependencies are part of our logical database design. We must decide which FD's should hold, which is a design choice.

	$\leftarrow$ A's $\rightarrow$	$\leftarrow$ B's $\rightarrow$	
t			
u			
	If t and u agree here...	Then t and u must agree here.	

Table 6: The Effect of A Functional Dependency On Two Tuples

2.2 Closure and Keys

Lesson Scope

A key concept of the relational data model is the key of a relation. The values of a key are unique in any possible relation instance and the attributes of a key can be inferred directly from the set of functional dependencies. A number of identities / rules are defined for functional dependencies and this lesson covers how we use them to determine the closure of a set of attributes and thereby whether or not that set of attributes is a key. This lesson covers §3.1.2–3.2.6 of the Garcia-Molina (2007) text.

Lesson Objectives

1. When asked, you can formally define a key and a superkey.
2. Given an attribute set A and a set of functional dependencies F, you can identify the closure of A.
3. Given a relation R and a set of functional dependencies F, you can identify all keys and superkeys of R.

Delivery Method

This lesson is focused on learning mathematical techniques which are components of more difficult mathematical problems that we will encounter later in this module. We will walk through a slide deck that introduces formal, mathematical definitions and works through several examples of solving problems per those definitions. It will follow the method of Explicit Modeling, in which we first see each problem decomposed into a series of steps, then the instructor will work through examples to demonstrate process, and finally we will try to solve some examples together in the manner just demoed.

Preparation

You are encouraged to read the sections of the textbook in advance, but this is not required to follow the lecture.

2.2.1 Functional Dependencies: Logical Rules

We know that functional dependencies are logical inferences so we then apply a number of logic rules to expand our set of FD's.

Combining Rule Given two rules with a common antecedent, we can combine them by taking the union of their consequents.

Given the two rules $A \rightarrow B$ and $A \rightarrow C$ we can combine them to produce a FD with a multi attribute consequent: $A \rightarrow BC$.

Splitting Rule Given a FD with a multi attribute consequent, we can split it into a series of rules in which the antecedent can functionally determine any subset of the consequent. (Thus, the inverse of the combining rule).

Given the multi attribute rule $A \rightarrow BC$ we can use the splitting rule to produce two new FD's with a singleton consequents: $A \rightarrow B$ and $A \rightarrow C$.

Transitivity Given two FD's of the form $A \rightarrow B$ and $B \rightarrow C$, we can infer that $A \rightarrow C$.

Triviality We can always assert a trivial FD of the form $A \rightarrow B$ if $B \subseteq A$.

If t and u agree on the A's... The they must agree on the B's... So surely they agree on the C's.

Semi-Triviality Given an FD of the form $A \rightarrow B$ and $C = B \cap A$, we can drop C from B.

2.2.2 The Closure of An Attribute Set

Given a set of attributes X, what is the maximum set of attributes that we can functionally determine form them? This is called the closure of X, and is denoted X^+. Figure 27 illustrates the methodology of closure.

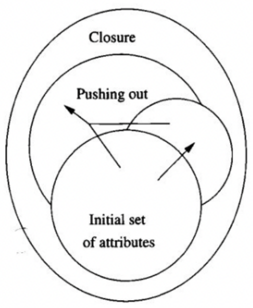


Figure 27: Computing the Closure of a Set of Attributes

Starting with the given set of attributes, we repeatedly expand the set by adding the right sides of FD's as soon as we have included their left sides. Eventually, we cannot expand the set any further, and the resulting set is the closure.

2.2.3 Computing the Closure of Attributes

Algorithm: Closure of a Set of Attributes.
INPUT: A set of attributes $\{A_1, A_2, ..., A_n\}$ and a set of FD's S.
OUTPUT: The closure $\{A_1, A_2, ..., A_n\}^+$.

1. If necessary, split the FD's of S, so each FD in S has a single attribute on the right.
2. Let X be a set of attributes that eventually will become the closure. Initialize X to be $\{A_1, A_2, ..., A_n\}$.
3. Repeatedly search for some FD

$$B_1, B_2, ..., B_m \rightarrow C$$

   such that all of $B_1, B_2, ..., B_m$ are in the set of attributes X, but C is not. Add C to the set X and repeat the search. Since X can only grow, and the number of attributes of any relation schema must be finite, eventually nothing more can be added to X, and this step ends.
4. The set X, after no more attributes can be added to it, is the correct value of

$$\{A_1, A_2, ..., A_n\}^+. \ \square$$

Example

Consider that you have the following FD's:

$A \rightarrow C$
$BC \rightarrow D$
$C \rightarrow B$

What is the closure of $\{A\}$, i.e., what is $\{A\}^+$?

1. Observe that $A \rightarrow C$; so $\{A\}^+ \supseteq \{A, C\}$.
2. Observe that $C \rightarrow B$; so $\{A\}^+ \supseteq \{A, B, C\}$.
3. Observe that $BC \rightarrow D$; so $\{A\}^+ \supseteq \{A, B, C, D\}$.
4. At this point, we cannot exploit any more rules; so, we conclude that $\{A\}^+ = \{A, B, C, D\}$.

What is the closure of $\{B\}$, i.e., what is $\{B\}^+$?

1. From triviality we have so $\{B\}^+ \supseteq \{B\}$.
2. But there are no FD's that use B in the antecedent; so, we conclude that $\{B\}^+ = \{B\}$.

2.2.4 (Super)key of A Relation

Let C be the set of attributes of relation R and $A \subseteq C$ be a subset of C.

A is a superkey if and only if:

1. $\{A\}^+ = C$.

Thus, every attribute of R can be functionally determined by A

Moreover, A is a key if and only if:

1. A is a superkey.
2. There is no set $A' \subset C$ for A' is a superkey.

In other words, A is minimal set of attributes from which the value of every tuple of R can be functionally determine; i.e., R is function from minimal domain A onto range C.

2.3 Minimal Bases and Projecting FD's

Lesson Scope

A set of functional dependencies (FD's) relate to a particular relation R. We might ask which of those functional dependencies would apply if we looked at a smaller relation that used only a subset of the attributes of R. This process of determining which FD's apply to the sub-relation is called "projection" and is more difficult than it might appear. Algorithmically, we determine it via brute force. This lesson covers §3.2.7–3.2.8 of the Garcia-Molina (2007) [2] text in which we learn how to calculate the set of FD's for a sub-relation, a technique that will be critical to the broader decomposition task that we will cover in the next two lessons.

Lesson Objectives

1. Given two sets of functional dependencies, F and F′, you can evaluate whether F′ is a basis for F.
2. Given a set of functional dependencies F, you can evaluate whether F is a minimal basis.
3. Given a relation R, a sub-relation R′ that has only some attributes of R, and a set of functional dependencies F, you can calculate the projection of F onto R′.

Delivery Method

This lesson is focused on learning mathematical techniques which are components of more difficult mathematical problems that we will encounter later in this module. We will walk through a slide deck that introduces formal, mathematical definitions and works through several examples of solving problems per those definitions. It will follow the method of Explicit Modeling, in which we first see each problem decomposed into a series of steps, then the instructor will work through examples to demonstrate process, and finally we will try to solve some examples together in the manner just demoed.

Preparation

You are encouraged to read the sections of the textbook in advance, but this is not required to follow the lecture.

2.3.1 Attribute Closures

In *relational design*, we specify:

- A relation schema, i.e., name and set of attributes.
- A set of functional dependencies, i.e., constraints that capture dependencies between attributes.

Given these, we can calculate the closure of any set of attributes using our rules of transitivity and (semi-)triviality.

2.3.2 Closure Of A Set Of Functional Dependencies

The closure of a set of functional dependencies (as opposed to a set of attributes) is:

- The closure of all possible subsets of the attributes.

Sometimes we have a choice of which FD's we use to represent the full set of FD's for a relation.

For example, given:
R(A, B, C)
$A \rightarrow B$
$C \rightarrow B$

The closure for $\{A \rightarrow B, C \rightarrow B\}$ is:

$\{\}^+ = \{\}$
$\{A\}^+ = \{A,B\}$
$\{B\}^+ = \{B\}$
$\{C\}^+ = \{B,C\}$
$\{A,B\}^+ = \{A,B\}$
$\{A,C\}^+ = \{A,B,C\}$
$\{B,C\}^+ = \{B,C\}$
$\{A,B,C\}^+ = \{A,B,C\}$

2.3.3 Basis Of A Set Of Functional Dependencies

We can then say that two sets of functional dependencies are equivalent if and only if:

- They have the same closing set.

When two sets of FD's are equivalent, we say they are a basis for each other.

Thus, given two sets of FD's, F and F′, we say that F′ is a basis for F (and vice versa) if and only if:

The closure in every subset of attributes is identical in F and F′.

We also say that F and F′ are equivalent.

We then ask: if it's clear that two sets of FD's can be equivalent, then we can ask if any of them are better than any others.

Example

Given:
R(A, B, C)
A $\rightarrow$ B
C $\rightarrow$ B

What other sets of FD's form a basis for this set {A $\rightarrow$ B, C $\rightarrow$ B}?
We can have the sets...

A $\rightarrow$ AB	A $\rightarrow$ B	AB $\rightarrow$ B
C $\rightarrow$ BC	B $\rightarrow$ B	A $\rightarrow$ AB
	C $\rightarrow$ B	C $\rightarrow$ B

2.3.4 Minimal Basis

Given a set of FD's F, we say that they form a minimal basis if and only if:

1. Every FD has a singleton right-hand side.
2. If any FD is removed, we no longer have a basis.
3. If we remove any attribute from the left-hand side of any FD, we no longer have a basis.

In other words, there is no simpler set of bases that form a basis for F.

2.3.5 Projecting Functional Dependencies

Given a set of FD's and a relation S with attributes C, we can produce a set of FD's for S by:

1. Calculate the closure of every non-empty subset c of C to create an FD: $c \rightarrow c^+$.
2. Simplify the resulting set of $2^{|C|} - 1$ FD's into a minimal basis.

Example

Let's project R(A, B, C, D, E) onto S(A, B, C) and then project the following FD's onto S as well:

A $\rightarrow$ B
B $\rightarrow$ D
CD $\rightarrow$ E

We start by listing all the functional dependencies...

Since there may be a large number of such FD's, and many of them may be redundant (i.e., they follow from other such FD's), we are free to simplify that set of FD's.

A → ABD
B → BD
AB → ABD
AC → ABCDE
AD → ABD
AE → ABDE
BC → BCD
BD → BD
BE → BDE
CD → CDE
CE → CE
DE → DE
ABC → ABCDE
ABD → ABD
ABE → ABDE
ACD → ABCDE
ACE → ABCDE
ADE → ABDE
BCD → BCDE
BCE → BCDE
BDE → BDE
CDE → CDE
ABCD → ABCDE
ABCE → ABCDE
ABDE → ABDE
ACDE → ABCDE
BCDE → BCDE

We start by removing all trivial and semi-trivial functional dependencies.

A → ~~A~~BD
B → ~~B~~D
AB → ~~AB~~D
AC → ~~A~~B~~C~~DE
AD → ~~A~~B~~D~~
AE → ~~A~~BD~~E~~
BC → ~~BC~~D
~~BD~~ → ~~BD~~
BE → ~~B~~D~~E~~
CD → ~~CD~~E
~~CE~~ → ~~CE~~
~~DE~~ → ~~DE~~
ABC → ~~ABC~~DE
~~ABD~~ → ~~ABD~~
ABE → ~~AB~~D~~E~~
ACD → ~~A~~B~~CD~~E
ACE → ~~A~~B~~C~~D~~E~~
ADE → ~~A~~B~~DE~~
BCD → ~~BCD~~E
BCE → ~~BC~~D~~E~~
~~BDE~~ → ~~BDE~~
~~CDE~~ → ~~CDE~~
ABCD → ~~ABCD~~E
ABCE → ~~ABC~~D~~E~~
~~ABDE~~ → ~~ABDE~~
ACDE → ~~A~~B~~CDE~~
~~BCDE~~ → ~~BCDE~~

Remove everything not in S.

A → B~~D~~
~~B~~ → ~~D~~
~~AB~~ → ~~D~~
AC → B~~DE~~
~~AD~~ → ~~B~~
~~AE~~ → ~~BD~~
~~BC~~ → ~~D~~
~~BE~~ → ~~D~~
~~CD~~ → ~~E~~
~~ABC~~ → ~~DE~~
~~ABE~~ → ~~D~~
~~ACD~~ → ~~BE~~
~~ACE~~ → ~~BD~~
~~ADE~~ → ~~B~~
~~BCD~~ → ~~E~~
~~BCE~~ → ~~D~~
~~ABCD~~ → ~~E~~
~~ABCE~~ → ~~D~~
~~ACDE~~ → ~~B~~

Reduce to a minimal basis.

A → B
AC → B

We can remove the second FD and still have a basis. Therefore the FD's for S are:

F′: A → B

Notice that we are able to simplify our process by not calculating any functional dependencies with attributes not in S.

2.3.6 Projecting a Set of Functional Dependencies

Algorithm: Projecting a Set of Functional Dependencies
INPUT: A relation R and a second relation R_1 computed by the projection $R_1 = \pi_L(R)$. Also, a set of FD's S that hold in R.
OUTPUT: The set of FD's that hold in R_1.
METHOD:

1. Let T be the eventual output set of FD's. Initially, T is empty.
2. For each set of attributes X that is a subset of the attributes of R_1, compute X^+. This computation is performed with respect to the set of FD's S, and may involve attributes that are in the schema of R but not R_1. Add to T all nontrivial FD's $X \rightarrow A$ such that A is both in X^+ and an attribute of R_1.
3. Now, T is a basis for the FD's that hold in R_1, but may not be a minimal basis. We may construct a minimal basis by modifying T as follows:
   (a) If there is an FD F in T that follows from the other FD's in T, remove F from T.
   (b) Let $Y \rightarrow B$ be an FD in T, with at least two attributes in Y, and let Z be Y with one of its attributes removed. If $Z \rightarrow B$ follows from the FD's in T (including $Y \rightarrow B$), then replace $Y \rightarrow B$ by $Z \rightarrow B$.
   (c) Repeat the above steps in all possible ways until no more changes to T can be made.

□

2.4 Boyce-Codd Normal Form (BCNF)

Lesson Scope

Certain types of data quality errors, also known as anomalies, cannot occur with a well-designed relational schema. More specifically, a relational schema is normalised if all its functional dependencies meet a particular condition. There are several such "normal forms"; in this lesson, we look at Boyce-Codd Normal Form (BCNF). A relational schema normalised in BCNF guarantees by design that insertion, update, and deletion anomalies are impossible. This lesson covers §3.3 of the Garcia-Molina (2007) [2] text, which reviews the definition of BCNF and an algorithm for decomposing a relation into a set of sub-relations that are in BCNF.

Lesson Objectives

1. When asked, you can list and describe many of the anomalies that can arise in a relation that is not in BCNF.
2. When asked, you can formally define Boyce-Codd Normal Form (BCNF).
3. Given a relation R and a set of FD's, you can identify whether or not R is in BCNF.
4. Given a relation R and a set of FD's, you can decompose R into a set of sub-relations that are all in BCNF.

Delivery Method

This lesson is focused on learning mathematical techniques. We will walk through a slide deck that introduces formal, mathematical definitions and works through several examples of solving problems per those definitions. It will follow the method of Explicit Modeling, in which we first see each problem decomposed into a series of steps, then the instructor will work through examples to demonstrate process, and finally we will try to solve some examples together in the manner just demoed.

Preparation

It would be advantageous to read the relevant sections of the textbook in advance of class, but this is not expected. You should be comfortable with the following previous lessons:

- calculation of closures and keys.
- projection of functional dependencies.

2.4.1 Anomalies

Problems such as redundancy that occur when we try to cram too much into a single relation are called anomalies. The principal kinds of anomalies that we encounter are:

Redundancy Information is unnecessarily repeated.

Update Anomaly We may change information in one tuple but leave the same information unchanged in another.

Deletion Anomaly If a set of values becomes empty, we may lose other information as a side effect.

Example

We can see that we have a redundancy, an update, and an deletion anomaly in Table 7. The name Alice appears twice (redundancy), then if we updated Alice's name in CSC320 we would have an update error. Similarly, if Bob drops CSC370 then we may lose all instances of Bob.

v_number	name	class
V00111222	Alice	CSC370
V00111222	Alice	CSC320
V00333444	Bob	CSC370

Table 7: Relation Anomalies Example

Observe that decomposing R into (v_number, name) and (v_number, class) solves all three anomalies! We can then create two tables (using functional dependencies).

v_number	name
V00111222	Alice
V00333444	Bob

Table 8: Student Table

v_number	class
V00111222	CSC370
V00111222	CSC320
V00333444	CSC370

Table 9: Class Table

We can now update Alice's name in Table 8 without causing an update error since the redundancy has also been solved. Likewise, we can drop Bob in Table 9 from CSC370 without losing Bob's information.

2.4.2 Decomposing Relations

The accepted way to eliminate anomalies is to decompose relations. Consider that you have a relation R with attributes $A = \{A_1, A_2, ..., A_n\}$.

A *decomposition* of R splits it into two relations:

S with attributes $B = \{B_1, B_2, ..., B_m\}$ and T with attributes $C = \{C_1, C_2, ..., C_k\}$.

Such that:

- $A = B \cup C$
- $S = \pi_{(B_1, B_2, ..., B_m)}(R)$
- $T = \pi_{(C_1, C_2, ..., C_m)}(R)$

2.4.3 Un-Decomposing Relations

To get R back...

Observe: BCNF decomposition partitions attributes into three sets:

- A key for relation S (A's)
- Determined by A's (B's)
- Not-determined (Others)

We can join each tuple of T to a unique tuple of S on the common A's. **But** A's must be in both relations. **And** a key for one of them.

2.4.4 Boyce-Codd Normal Form

The goal of decomposition is to obtain a set of relations that are all in some normal form. A relation R with attribute set C is in BCNF if and only if:

For every non-trivial functional dependency on R, $\{A_1, A_2, ..., A_n\} \rightarrow \{B_1, B_2, ..., B_m\}$, $\{A_1, A_2, ..., A_n\}$ is a superkey of R (i.e., $A^+ = C$).

Stated informally, no relation has a functional dependency in which the antecedent does not determine every other attribute. We note that this requires a superkey, not necessarily a key.

2.4.5 BCNF Decomposition Algorithm

In general, we must keep applying the decomposition rule as many times as needed, until all our relations are in BCNF. We can be sure of ultimate success, because every time we apply the decomposition rule to a relation R, the two resulting schemas each have fewer attributes than that of R.

BCNF_Decomposition(R_0, F_0**)**
Input: Relation R_0 with attribute set C_0 and functional dependencies F_0.
Output: A decomposition of R_0 in which all relations are in BCNF.

1. If R_0 is in BCNF, return R_0.
2. Select a BCNF violation (an FD for which the antecedent is not a superkey), $X \to Y$.
3. Compute the closure X^+.
4. Let $R_1 := X^+$.
5. Let $R_2 := (C \setminus X^+) \cup X$.
6. Project F_0 to get functional dependencies for R_1 and R_2 denoted F_1 and F_2, respectively.
7. Return BCNF_Decomposition(R_1, F_1) $\cup$ BCNF_Decomposition(R_2, F_2).

2.5 Third Normal Form (3NF)

Lesson Scope

While BCNF is excellent at eliminating anomalies, no normal form is perfect. In this lesson, we will see an example where BCNF decomposition leads to a loss of some design characteristics, namely some functional dependencies are lost. We learn an alternative normal form, Third Normal Form (3NF), and an alternative algorithm to decompose a relation R into 3NF. This lesson covers §3.4.4 and §3.5 of the Garcia-Molina (2007) [2] text.

Lesson Objectives

1. When asked, you can provide an example relation R and set of FD's for which BCNF decomposition will lose at least one FD.
2. When asked, you can formally define prime attributes and Third Normal Form (3NF).
3. Given a relation R and a set of FD's, you can identify whether or not R is in 3NF.
4. Given a relation R and a set of FD's, you can decompose R into a set of sub-relations that are all in 3NF.

Delivery Method

This lesson is focused on learning mathematical techniques. We will walk through a slide deck that introduces formal, mathematical definitions and works through several examples of solving problems per those definitions. It will follow the method of Explicit Modeling, in which we first see each problem decomposed into a series of steps, then the instructor will work through examples to demonstrate process, and finally we will try to solve some examples together in the manner just demoed.

Preparation

It would be advantageous to read the relevant sections of the textbook in advance of class, but this is not expected. You should be comfortable with the following previous lessons:

- calculation of closures and keys.
- projection of functional dependencies.

2.5.1 Definition of Third Normal Form

A relation R over attributes C with functional dependencies F is in 3NF if and only if:

Every functional dependency f in F is in 3NF.

A functional dependency $f : X \rightarrow Y$ is in 3NF in and only if:

X is a superkey OR Every attribute in Y is "prime".

Equivalently:

f is in BCNF OR Every attribute in Y is "prime".

An attribute that is a member of some key is often said to be prime. Thus, the 3NF condition can be stated as "for each nontrivial FD, either the left side is a superkey, or the right side consists of prime attributes only."

Note that the difference between this 3NF condition and the BCNF condition is the clause "is a member of some key (i.e., prime)."

2.5.2 The Synthesis Algorithm for 3NF Schemas

We can now explain and justify how we decompose a relation R into a set of relations such that:

(a) The relations of the decomposition are all in 3NF.

(b) The decomposition has a lossless join.

(c) The decomposition has the dependency-preservation property.

Algorithm: Synthesis of Third-Normal-Form Relations With a Lossless Join and Dependency Preservation.
INPUT: A relation R and a set F of functional dependencies that hold for R.
OUTPUT: A decomposition of R into a collection of relations, each of which is in 3NF. The decomposition has the lossless-join and dependency-preservation properties.
METHOD: Perform the following steps:

1. Find a minimal basis for F, say G.
2. For each functional dependency $X \rightarrow A$ in G, use XA as the schema of one of the relations in the decomposition.
3. If none of the relation schemas from Step 2 is a superkey for R, add another relation whose schema is a key for R. □

2.5.3 Normal Forms

There are several other normal forms that we have not looked at:

- First Normal Form: Every component of every tuple is an atomic value.
- Second Normal Form: A less restrictive version of 3NF.
- Fourth Normal Form: Is essentially the BCNF condition, but applied to multi variables instead of functional dependencies.

We can construct a containment hierarchy, as seen in Figure 28, of the different normal forms.

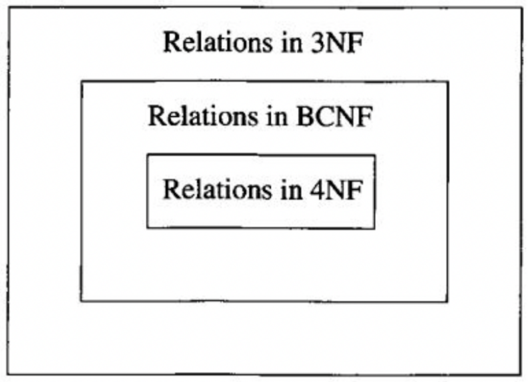


Figure 28: 4NF implies BCNF implies 3NF

2.5.4 Normal Form Summary

There are a number of different normal forms to reduce/prevent these anomalies, each with different advantages...

- 3NF preserves all the functional dependencies that we have identified.
- BCNF eliminates redundancy due to FD's.

We note that both decomposition algorithms are non-deterministic. And after normalisation, we have completed the logical design

2.6 Converting ERD's into Relational Schemata

Lesson Scope

This module so far has presented logical design per the relational model as a stand-alone process. We may recall from the first module, however, that one often starts with a conceptual design, say as an Entity-Relationship Diagram (ERD), when designing a logical schema. This lesson covers §4.5–4.6 of the Garcia-Molina (2007) text, describing how to convert an entity-relationship diagram into a set of relations and a set of relational algebra constraints, including referential integrity, key, and degree constraints. Reconsidering the conceptual design quality of "normalised," we also look at how a good conceptual design naturally leads to a BCNF-normalised logical design. It therefore ties together the first and second modules of the course.

Lesson Objectives

1. Given a conceptual design expressed as an Entity-Relationship Diagram (ERD), you can design a set of relations and relational algebra constraints that fully and precisely captures the conceptual design.
2. Given a natural language description of a key, referential integrity, or degree constraint C, you can write C as a relational algebra constraint.
3. Given an ERD, you can identify all functional dependencies and then use BCNF decomposition to verify that the ERD has the design quality of normalisation.

Delivery Method

This lesson is focused on learning mathematical techniques. We will walk through a slide deck that introduces formal, mathematical definitions and works through several examples of solving problems per those definitions. It will follow the method of Explicit Modeling, in which we first see each problem decomposed into a series of steps, then the instructor will work through examples to demonstrate process, and finally we will try to solve some examples together in the manner just demoed.

Preparation

It would be advantageous to read the relevant sections of the textbook in advance of class, but this is not expected. You should be comfortable with the following previous lessons:

- Interpreting Entity-Relationship Diagrams (ERD's).
- The syntax of constraints in relational algebra.
- BCNF decomposition.

2.6.1 Converting ERD's

To a first approximation, converting an E/R design to a relational database schema is straightforward:

- Turn each entity set into a relation with the same set of attributes, and
- Replace a relationship by a relation whose attributes are the keys for the connected entity sets.

While these two rules cover much of the ground, there are also several special situations that we need to deal with, including:

1. Weak entity sets cannot be translated straightforwardly to relations.
2. "Isa" relationships and subclasses require careful treatment.
3. Sometimes, we do well to combine two relations, especially the relation for an entity set E and the relation that comes from a many-one relationship from E to some other entity set.

2.6.2 Entity Sets

Each entity set can be converted into a relation. Identifiers functionally determine non-identifiers in the same entity set.

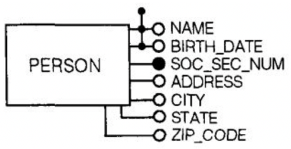


Figure 29: Entity Person

We can see how to do this conversion by using Figure 29. We obtain the following relation and functional dependencies:

Person(name, birthdate, ssn, address, city, state, zip)

ssn $\rightarrow$ name, birthdate, address, city, state, zip
name birthdate $\rightarrow$ ssn

The functional dependencies show that ssn can determine all attributes and that name and birthdate can determine ssn.

2.6.3 Relationships

Relationships in the E/R model are also represented by relations. The relation for a given relationship R has the following attributes:

1. For each entity set involved in relationship R, we take its key attribute or attributes as part of the schema of the relation for R.
2. If the relationship has attributes, then these are also attributes of relation R.

If one entity set is involved several times in a relationship, in different roles, then its key attributes each appear as many times as there are roles. We must rename the attributes to avoid name duplication. More generally, should the same attribute name appear twice or more among the attributes of R itself and the keys of the entity sets involved in relationship R, then we need to rename to avoid duplication.

2.6.4 Relationships with Attributes

Each entity set can be converted into a relation. Attributes go where the relationships goes.

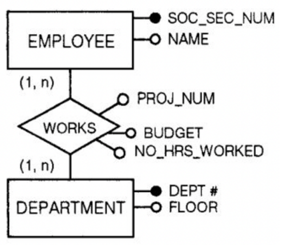


Figure 30: Employee and Department Works Relationship with Attributes

The works relationship in Figure 30 has 3 attributes. Thus, we can get the following relations:

Employee(<u>ssn</u>, name)
Department(<u>deptNum</u>, floor)
Works(<u>ssn</u>, <u>deptNum</u>, projectNum, budget, hours)

ssn $\rightarrow$ name
deptNum $\rightarrow$ floor
ssn deptNum $\rightarrow$ projectNum budget hours

We note that the works relation incorporates identifiers from employee and department.

2.6.5 Relationships - Many-One

Each entity set can be converted into a relation. The relationship is exactly what we got from a typical BCNF decomposition

Figure 31 shows a many-one relationship with employee and director. An employee can only work with one director, but a director can work with many employees. So we get the following relations and functional dependencies:

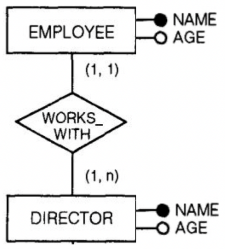


Figure 31: Employee and Director Works With Relationship

Employee(<u>name</u>, age)
Director(<u>name</u>, age)

employee.name → employee.age director.name
director.name → director.age

And so we can then join the works with relationship with the employee relation to obtain the following:

Employee(<u>name</u>, age, WORKS_WITH)

We also note that we can use director.name instead of works with but there are advantages to using works with relation.

2.6.6 Relationships - Many-Many

Each entity set can be converted into a relation. The relationship is like what we got from a BCNF decomposition with independent FD's.

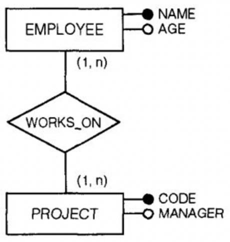


Figure 32: Employee and Project Works On Relationship

In Figure 32 we have a relationship works on between employee and project that is many-many. We obtain the following relations and functional dependencies:

Employee(<u>name</u>, age)
Project(<u>code</u>, manager)

employee.name → age
project.code → manager

We can then relate the two with a WORKS_ON relation:

WorksOn(<u>name</u>, <u>code</u>)

Here we have a separate relation to handle to many-many relationship.

2.6.7 Relationships - One-One

Can be enforced with constraints. Or, can move one entire entity set into another entity set.

2.6.8 Subsets

Subsets are like other entity sets. They share the same key.

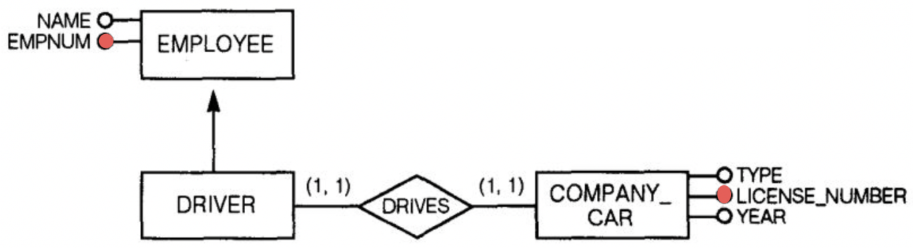


Figure 33: Employee, Driver, and Company Car Drives Relationship

The drives relationship shown in Figure 33 has a subset entity set driver from employee. We can observe the following relations and functional dependencies:

Employee(<u>employeeNum</u>, name)
Driver(<u>employeeNum</u>)
CompanyCar(<u>licenseNum</u>, type, year)

employeeNum $\rightarrow$ name licenseNum
licenseNum $\rightarrow$ type year employeeNum

We observe the symmetry in the FD's. We also note that we can remove driver and have the following instead:

~~**Driver**(employeeNum)~~
CompanyCar(<u>licenseNum</u>, type, year, driver)

We can add driver to the relation company car as driver inherits employeeNum.

2.6.9 Multi-Level Subsets

Subsets are like other entity sets. They share the same key. Always refer directly to parent.

When we have subsets, such as in Figure 34, we can see that computer expert and analyst have the same relations. Thus, we get the following relations:

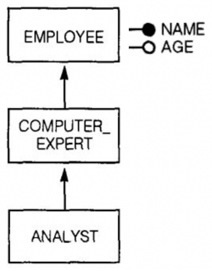


Figure 34: Employee Milti Subset Hierarchy with Computer Expert and Analyst

Employee(name, age)
ComputerExpert(name)
Analyst(name)

In this case we have dangling entities, but this applies to all cases.

2.6.10 Weak Identity Sets

When a weak entity set appears in an E/R diagram, we need to do three things differently.

1. The relation for the weak entity set W itself must include not only the attributes of W but also the key attributes of the supporting entity sets. The supporting entity sets are easily recognized because they are reached by supporting relationships from W.
2. The relation for any relationship in which the weak entity set W appears must use as a key for W all of its key attributes, including those of other entity sets that contribute to W's key.
3. However, a supporting relationship R, from the weak entity set W to a supporting entity set, need not be converted to a relation at all. The justification is that the attributes of many-one relationship R's relation will either be attributes of the relation for W, or (in the case of attributes on R) can be added to the schema for W's relation.

Of course, when introducing additional attributes to build the key of a weak entity set, we must be careful not to use the same name twice. If necessary, we rename some or all of these attributes.

Alternatively the following is a modified rule for converting to relations entity sets that are weak.

- If W is a weak entity set, construct for W a relation whose schema consists of:
  1. All attributes of W.
  2. All attributes of supporting relationships for W.
  3. For each supporting relationship for W, say a many-one relationship from W to entity set E, all the key attributes of E.

Rename attributes, if necessary, to avoid name conflicts.

- Do not construct a relation for any supporting relationship for W.

2.7 Relational Algebra and Constraints

Lesson Scope

Until this point, we have considered two types of constraints in the relational model: key (i.e., uniqueness) constraints and functional dependencies. The data model also includes an algebraic language, called relational algebra, for expressing operations and constraints of arbitrary complexity. This lesson covers §2.4–2.5 of the Garcia-Molina (2007) [2] text, and will introduce the operators of the relational data model, relational algebra syntax, and the form that constraints take when expressed in that syntax.

Lesson Objectives

1. Given relation instances and a relational operator applied to them, you can derive the exact relation instance that is output by the operator.
2. Given a constraint expressed in relational algebra, you can articulate precisely in plain English what the constraint prevents.
3. Given a constraint expressed in plain English, you can correctly write an equivalent expression in relational algebra.

Delivery Method

This lesson is focused on learning mathematical techniques. We will walk through a slide deck that introduces formal, mathematical definitions and works through several examples of solving problems per those definitions. It will follow the method of Explicit Modeling, in which we first see each problem decomposed into a series of steps, then the instructor will work through examples to demonstrate process, and finally we will try to solve some examples together in the manner just demoed.

Preparation

It would be advantageous to read the relevant sections of the textbook in advance of class, but this is not expected. If you are rusty on the basics of predicate logic and universal/existential quantifiers, you may benefit from reviewing these two eight-minute videos, as this knowledge will be assumed during the lesson:

- Universal and Existential Quantifiers
- Negating Universal and Existential Quantifiers

2.7.1 Rename Operator

The rename operator (ρ) takes a relation as input and maps it onto a new schema of the same size, i.e., renames the attributes. This will allow us to apply set theoretic operators to different schemata. If we only want to change the name of the relation to S and leave the attributes as they are in R, we can just say $\rho_{\text{S}}(\text{R})$.

Example Given R(A, B, C, D) we can obtain S(B, C, D, E) by applying:

$$\rho_{\text{S(B, C, D, E)}}(\text{R})$$

Which maps R.A onto S.B, R.B onto S.C, R.C onto S.D, and R.D onto S.E.

A	B	C	D
a	b	c	d

Table 10: R

B	C	D	E
a	b	c	d

Table 11: S

2.7.2 Set-Theoretic Operators

When we apply these operations to relations, we need to put some conditions on R and S:

1. R and S must have schemas with identical sets of attributes, and the types (domains) for each attribute must be the same in R and S.
2. Before we compute the set-theoretic union, intersection, or difference of sets of tuples, the columns of R and S must be ordered so that the order of attributes is the same for both relations.

2.7.3 Set-Theoretic Operators: Union

The union ($\cup$) takes two relations as input and retains any tuples that appears in either, i.e., calculates a set union.

R $\cup$ S, the union of R and S, is the set of elements that are in R or S or both. An element appears only once in the union even if it is present in both R and S.

Example R $\cup$ $\rho_{\text{S(X,Y)}}(\text{S})$ first applies a renaming to S so that R and S contain tuples from the same schema, then takes a set union.

X	Y
x1	y1

Table 12: R

X	Z
x1	z1
x1	y1

Table 13: S

X	Y
x1	z1
x1	y1

Table 14: R′

2.7.4 Set-Theoretic Operators: Intersection

The intersection $\cap$ takes two relations as input and retains any tuples that appears in both, i.e., calculates a set intersection.

R $\cap$ S, the intersection of R and S, is the set of elements that are in both R and S.

Example $R \cap \rho_{S(X,Y)}(S)$ first applies a renaming to S so that R and S contain tuples from the same schema, then takes a set intersection.

X	Y
x1	y1

Table 15: R

X	Z
x1	z1
x1	y1

Table 16: S

X	Y
x1	y1

Table 17: R′

2.7.5 Set-Theoretic Operators: Difference

The difference $(-)$ takes two relations as input and retains any tuples that only appears in the first, i.e., calculates a set difference.

R $-$ S, the difference of R and S, is the set of elements that are in R but not in S. Note that R $-$ S is different from S $-$ R; the latter is the set of elements that are in S but not in R.

Example $R - \rho_{S(X,Y)}(S)$ produces the empty set, $\emptyset$. It still has schema R′(X,Y).

X	Y
x1	y1

Table 18: R

X	Z
x1	z1
x1	y1

Table 19: S

X	Y

Table 20: R′

2.7.6 Filter Operators: Projection

The projection (π) takes a relation as input and retains only a subset of the relation schema, i.e., drops some attributes.

Example 1 Given R(A, B, C, D) applying $\pi_{(B,\ C)}(R)$ drops attributes A and D from R.

A	B	C	D
a1	b1	c1	d1
a2	b2	c2	d2

Table 21: R

B	C
b1	c1
b2	c2

Table 22: R′

Example 2 Given R(A, B, C, D). Observe that the two tuples are unique because of attribute A. Applying $\pi_{(B, C)}(R)$ eliminates a tuple because the disambiguation column A was dropped.

A	B	C	D
a1	b	c	d
a2	b	c	d

Table 23: R

B	C
b	c

Table 24: R′

2.7.7 Filter Operators: Selection

The selection (σ) takes a relation as input and retains only those tuples that match a predicate, i.e., drops some attributes.

C is a conditional expression. The operands in condition C are either constants or attributes of R.

Example 1 Given R(A, B, C, D) applying $\sigma_{A = a1}(R)$ drops the second row because a2 != a1.

A	B	C	D
a1	b1	c1	d1
a2	b2	c2	d2

Table 25: R

A	B	C	D
a1	b1	c1	d1

Table 26: R′

Example 2 Given R(A, B, C, D) applying $\sigma_{(X != b1)}(\rho_{(S(X, Y))}(\pi_{(B, C)}(R)))$ projects R, renames it, then drops tuples for which X = b1.

A	B	C	D
a1	b1	c1	d1
a2	b2	c2	d2

Table 27: R

X	Y
b2	c2

Table 28: R′

2.7.8 Cross Product Operator

The cross product (or Cartesian) operator ($\times$) takes two relations as input and combines every tuple pairwise.

The relation schema for the resulting relation is the union of the schemas for R and S. However, if R and S should happen to have some attributes in common, then we need to invent new names for at least one of each pair of identical attributes. To disambiguate an attribute A that is in the schemas of both R and S, we use R.A for the attribute from R and S.A for the attribute from S.

Example R × S merges each row of R with each row of S. Note the forced renaming.

X	Y
x1	y1
x2	y2

Table 29: R

X	Z
x1	z1
x1	y1

Table 30: S

R.X	Y	S.X	Z
x1	y1	x1	z1
x1	y1	x1	y1
x2	y2	x1	z1
x2	y2	x1	y1

Table 31: R′

2.7.9 Natural Join Operator

The natural join operator ($\bowtie$) takes two relations as input and combines only those tuples that agree on common columns.

More precisely, let $A_1, A_2, ..., A_n$ be all the attributes that are in both the schema of R and the schema of S. Then a tuple r from R and a tuple s from S are successfully paired if and only if r and s agree on each of the attributes $A_1, A_2, ..., A_n$.

If the tuples r and s are successfully paired in the join R $\bowtie$ S, then the result of the pairing is a tuple, called the joined tuple, with one component for each of the attributes in the union of the schemas of R and S.

Example R $\bowtie$ S merges each tuple of R and tuple of S that have the same value for X. This is in fact exactly the same as: $\rho_{(\text{X, Y, Z})}(\pi_{(\text{R.X, Y, Z})}(\sigma_{\text{R.X = S.X}}(\text{R} \times \text{S})))$.

X	Y
x1	y1
x2	y2

Table 32: R

X	Z
x1	z1
x1	y1

Table 33: S

X	Y	Z
x1	y1	z1
x1	y1	y1

Table 34: R′

2.7.10 Theta Join Operator

The theta join operator ($\bowtie_{\text{Condition}}$) takes two relations as input and combines only those tuples for which the specified condition is true.

The notation for a theta join of relations R and S based on condition C is R $\bowtie_{\text{C}}$ S. The result of this operation is constructed as follows:

1. Take the product of R and S.
2. Select from the product only those tuples that satisfy the condition C.

As with the product operation, the schema for the result is the union of the schemas of R and S, with "R" or "S". prefixed to attributes if necessary to indicate from which schema the attribute came.

Example R $\bowtie_{\text{R.X = X.S AND R.Y != S.Z}}$ S merges the only pair of tuples r ∈ R and s ∈ S for which the condition is true.

X	Y
x1	y1
x2	y2

Table 35: R

X	Z
x1	z1
x1	y1

Table 36: S

R.X	Y	Z	S.X
x1	y1	z1	x1

Table 37: R′

2.7.11 Constraint

A constraint is a limitation on what data can be. Ensuring that the constraints of a database are maintained is our best defense against the deterioration of data quality.

There are two ways in which we can use expressions of relational algebra to express constraints.

1. If R is an expression of relational algebra, then R = ∅ is a constraint that says "The value of R must be empty", or equivalently "There are no tuples in the result of R".
2. If R and S are expressions of relational algebra, then R ⊆ S is a constraint that says "Every tuple in the result of R must also be in the result of S". Of course the result of S may contain additional tuples not produced by R.

These ways of expressing constraints are actually equivalent in what they can express, but sometimes one or the other is clearer or more succinct. That is, the constraint R ⊆ S could just as well have been written R − S = ∅. To see why, notice that if every tuple in R is also in S, then surely R − S is empty. Conversely, if R − S contains no tuples, then every tuple in R must be in S (or else it would be in R − S).

On the other hand, a constraint of the first form, R = ∅, could just as well have been written R ⊆ ∅. Technically, ∅ is not an expression of relational algebra, but since there are expressions that evaluate to ∅, such as R − R, there is no harm in using ∅ as a relational-algebra expression.

2.7.12 Referential Integrity Constraints

A common kind of constraint, called a referential integrity constraint, asserts that a value appearing in one context also appears in another, related context.

In general, if we have any value v as the component in attribute A of some tuple in one relation R, then because of our design intentions we may expect that v will appear in a particular component (say for attribute B) of some tuple of another relation S. We can express this integrity constraint in relational algebra as:

$$\pi_A(R) \subseteq \pi_B(S)$$

or equivalently,

$$\pi_A(R) - \pi_B(S) = \emptyset.$$

2.7.13 Key Constraints

The same constraint notation allows us to express far more than referential integrity. Here, we shall see how we can express algebraically the constraint that a certain attribute or set of attributes is a key for a relation.

$$\rho_A(R) \bowtie_{A.a=B.a \wedge (A.b \neq B.b \vee A.c \neq B.c)} \rho_B(R) = \emptyset$$

2.7.14 Additional Constraint Examples

There are many other kinds of constraints that we can express in relational algebra and that are useful for restricting database contents. A large family of constraints involve the permitted values in a context. For example, domain constraints for an attribute.

2.8 Midterm 02

2.8.1 Scope

This exam covers the logical design unit of the course, including Chapters 2 and 3 and §4.5–4.6 of the Garcia-Molina et al. textbook. This overlaps all class lectures strictly later than Monday, 26 September and strictly earlier than Monday, 17 October. The exam assesses your knowledge of the relational data model, particularly that:

- when describing aspects of a relational database, you can use the correct terminology.
- given a relation R and a set of functional dependencies F, you can decompose R into BCNF or 3NF.
- given an Entity-Relationship Diagram (ERD), you can convert it into a relational schema, including relational algebra constraints.
- given a conceptual schema, you can translate it precisely into a normalised relational database schema.

2.8.2 Format

This *closed-book, multiple choice* exam is written in-person with responses recorded on a standard five-option UVic bubble sheet. You may bring to the exam one single-sided A4 or US Letter page of hand-written or typed notes and blank scrap paper. The practice exam below is highly predictive of the style of questions that you can expect. You have up to forty-five minutes to write the exam, after which we will take a ten minute break and then resume with a shortened lecture. The exam will not be live-streamed over Zoom nor recorded, but the lecture thereafter will be.

For online-first students (A03), please note that there have been sufficiently many course withdrawals and CAL registrations that there is space for everyone to write the exam in the lecture theatre, even without considering that some students may choose to only write the assignment. Please feel free to join us if you want.

3 Structured Query Language (SQL) and Transactions

Overview

Structured Query Language (SQL), pronounced "sequel" or "ess-queue-ell," is the standard declarative programming language for interacting with a relational database. It contains syntax for:

- defining a relational database per the "data definition language" (DDL).
- modifying data within the database per the "data manipulation language" (DML).
- extracting information from the database per the "data query language" (DQL).

Moreover, SQL supports batching queries together as transactions so that they are executed as an atomic unit, maintain the consistency of the database, permit specifying an appropriate isolation level in the midst of concurrent use, and guarantee the durability of the data—the so-called ACID properties of a database.

SQL also permits specifying conditional behaviour so that, in the event one things happens in the database, something else is triggered to happen as well.

In this module, we will learn all these aspects of the SQL programming language so that you can interact declaratively with the main features of a relational database management system.

Module-Level Intended Learning Outcomes

Upon completion of this short introductory module, you can:

- given a logical database model, construct a series of SQL DDL queries that will fully implement the structure and constraints of the model.
- given a relational schema and an information need, construct a SQL DQL query to accurately retrieve that information.
- given a relational schema and a series of data updates that need to be performed, construct a SQL transaction to correctly execute those updates.
- given a scenario involving a database, describe how it impacts the ACID properties of an RDBMS.
- given a set of transactions, describe how various isolation levels for those transactions impact concurrent execution.

References

Garcia-Molina, Ullman, and Widom (2009) [2]. Database Systems: The Complete Book. Pearson Prentice Hall. Chapters 5–8.

3.1 The SQL Data Definition Language (DDL)

Lesson Scope

The primary programming language for interacting with a relational database management system (RDBMS) is the structured query language, SQL (pronounced by some as "ess-queue-ell" and by others as "sequel"). SQL uses the declarative programming paradigm, which you have not likely encountered in previous classes. Declarative languages are very high-level languages that specified what should be done but not how. SQL has two components: the data definition language (DDL) for specifying a database design that should be created; and the data manipulation language (DML) for interacting with the data in table instances.

This lesson covers §2.3 of the Garcia-Molina (2007) text, introducing the DDL component of SQL. We will learn how to implement a relational schema using SQL. It therefore serves as a bridge to the third module of the course which is focused on SQL.

Lesson Objectives

1. Given a relational schema, you can write syntactically and semantically correct SQL CREATE TABLE statements to build it as a MySQL database.
2. Given a relation and its key and referential integrity constraints, you can write syntactically and semantically correct SQL DDL statements to implement primary key and foreign key constraints in an existing MySQL database.
3. Given a modification to a relational schema and a MySQL database, you can write syntactically and semantically correct SQL DDL statements to reflect the modifications in the database.

Delivery Method

This lesson is focused on learning a new programming language. We will use a participatory form of Live Coding, where the instructor will design a database in front of the class, asking for recommendations and accepting ad-hoc design changes throughout. This is meant to focus more on the process of SQL programming under the assumption that the syntax is easy to understand for students who have taken the prerequisite courses.

Preparation

It would be advantageous to read the relevant sections of the textbook in advance of class, but this is not expected. You should be comfortable with the following previous lessons:

- The nomenclature of the relational data model.
- Key and referential integrity constraints.

3.1.1 Database Language

There are two components of a database language:

- Data Definition Language (DDL): Used to declare what schemata should exist in our database.
- Data Manipulation Language (DML): Used to declare what should be done to data inserted into an instance of that schemata (e.g., querying/ask questions and modifications).

SQL has syntax for both DDL and DML.

3.1.2 Types of Relations

A Table A stored relation (i.e., one that should be saved on some form of permanent storage like a disk).

A View An abstraction that is defined by some computation. We can consider it to be a Temporary Table generated by the query engine to save an intermediate result of a computation.

3.1.3 Creating A Database

Using MySQL on an Ubuntu 21 flavoured terminal...

```
sudo apt update
sudo apt install mysql_server
sudo mysql_secure_installation sudo mysql

create database [database-name];
use [database-name];
```

Replace what is in the square brackets ([]) with the appropriate text.

3.1.4 Creating A Table

To declare schema **Tress**(latinName, englishName, type) we execute (case insensitive):

```
CREATE TABLE Tress(latinName <data type>,
                   englishName <data type>,
                   type <data type>);
```

Listing 1: A Simple CREATE TABLE Example Statement

where <data type> is a built-in type defined as part of the SQL standard. However, this is not always the case.

Observe the symmetry between the schema and the create table statement. We only add the CREATE TABLE prefix, a semi-colon (;) to end the statement, and a data type to each attribute.

3.1.5 Data Types

Table 38 shows the available built-in data types in MySQL.

Name	Description
CHAR(n)	Fixed-length string of exactly n characters.
VARCHAR(n)	Variable-length (e.g., null-terminated) string of up to n characters.
BOOLEAN	Element from the set {TRUE, FALSE, UNKNOWN}.
INT or SHORTINT	Standard integer or one using fewer bits.
FLOAT or DOUBLE	Floating point value or one with double precision.
DATE	Special string of the format: YYYY-MM-DD.
TIME	Special string of the format: HH:MM:SS.millis.
ENUM	Custom-defined set that behaves like BOOLEAN.

Table 38: SQL Data Types

We can now review Listing 1 and apply the appropriate data types to create a syntactically correct statement.

```
CREATE TABLE Tress(latinName VARCHAR(20),
                   englishName VARCHAR(20),
                   type ENUM('deciduous', 'evergreen'));
```

Listing 2: Syntactically correct CREATE TABLE statement

Note: You can use whitespace however you like. Just keep in mind the person who has to read it. You can also redirect files of SQL code to MySQL using the following command: sudo mysql < build-trees.sql.

3.1.6 Adding Default Values In A Table

If we want to populate unspecified values with something other than the NULL symbol, we append a DEFAULT clause:

```
CREATE TABLE Trees(latinName VARCHAR(20)
                             DEFAULT 'lorem ipsum');
```

Listing 3: CREATE TABLE Statement with Default Value

3.1.7 Modifying A Table

To eliminate the table completely (including all its data and metadata!):
DROP TABLE Trees;

To simply alter the table (and preserve its contents):
ALTER TABLE Trees <the modification we want to make>;

To add a new attribute, we specify add with a data type, e.g.:
ALTER TABLE Trees ADD quantity INT;

To remove an attribute, we use the DROP command:
ALTER TABLE Trees DROP quantity;

To change the type (if it's possible),we use the MODIFY keyword exactly like the ADD modifier.

Example We can use the ALTER modifier to add default values:

```
ALTER TABLE Trees MODIFY type ENUM('deciduous', 'evergreen')
                              DEFAULT 'evergreen';
```

Listing 4: ALTER TABLE Statement with Default Value

3.1.8 Declaring the Key of A Relation

We can declare a primary key in one of two ways:

As a property of an attribute:

```
CREATE TABLE Trees(latinName VARCHAR(20) PRIMARY KEY);
```

Listing 5: CREATE TABLE with PRIMARY KEY Attribute

Or as a separate definition:

```
CREATE TABLE Trees(latinName VARCHAR(20),
                   englishName VARCHAR(20),
                   type ENUM('deciduous', 'evergreen'),
                   PRIMARY KEY(latinName));
```

Listing 6: CREATE TABLE with PRIMARY KEY Definition

3.1.9 Converting ERD's

Each entity set can be converted into a relation and implemented with a CREATE TABLE statement.

One-Many Relationship We can convert Figure 35 into SQL statements.

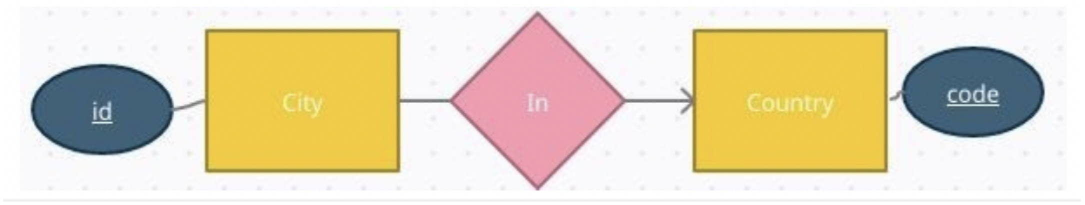


Figure 35: One-Many Relationship

We will begin by creating two relations:

```
CREATE TABLE City(id INT, PRIMARY KEY ('id'));
CREATE TABLE Country(code INT, PRIMARY KEY('code'));
```

Listing 7: CREATE TABLE City and Country

To add countryCode to our City relation we need to add a foreign key:

```
CREATE TABLE City(id INT,
                   country_code INT NOT NULL,
                   PRIMARY KEY(id),
                   FOREIGN KEY('country_code'),
                   REFERENCES Country('code'));
```

Listing 8: CREATE TABLE City and Country

Many-Many Relationship We can convert Figure into SQL statements.

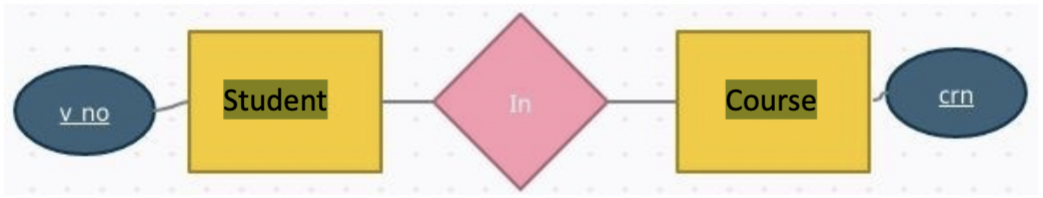


Figure 36: Many-Many Relationship

```
CREATE TABLE IN(student INT,
                course INT,
                PRIMARY KEY(student, course),
                FOREIGN KEY(student),
                REFERENCES Student(v_no),
                FOREIGN KEY(course),
        REFERENCES Course(crn));
```

Listing 9: CREATE TABLE IN

One-One Relationships We often model them as Many-One relationships.

3.1.10 Converting Relational Algebra

A cool website is Query Converter where it helps us convert query to the relational algebra.

3.2 Introduction to SQL

Lesson Scope

This lesson covers §6.1-6.2 of the textbook. It will cover SQL syntax for writing basic SELECT-FROM-WHERE DQL (data query language) queries, including all of the major operators: selection, projection, join, cross product, and duplicate elimination.

Lesson Objectives

1. Given a database schema and a basic query expressed in plain English or Datalog, you can formulate a correct SQL query with the same semantics.
2. Given a basic SELECT-FROM-WHERE clause, you can describe in plain English the effect of the query.

Delivery Method

We will visit together a number of basic queries in SQL.

Preparation

You are encouraged to read the sections of the textbook in advance, but this is not required to follow the lecture. You should be comfortable with converting a plain English statement into a declarative logic statement (such as in Datalog) so that we can focus today on syntax.

Lesson Materials

SQL code written during class.

Instructions

Download 'university_db_dump.sql' and create a new database called university. Add the dump file into the new database university.

Run the following in your terminal to import the university db:

```
mysql -u [username] -p -e "CREATE DATABASE university;"
mysql -u [username] -p university < university_db_dump.sql
```

Run the following in your sql terminal to start using the university db:

```
SHOW DATABASES;
USE university;
SHOW TABLES;
```

3.2.1 Basic SQL

We will be using the following DB schema:

Person(v_number, name)	Instructor(v_number, specialisation)
Student(v_number, major)	Class(code, name)
EnrolledIn(student, class, semester)	Teaches(instructor, class, semester)

SELECT, FROM, WHERE, and DISTINCT We will be demonstrating the foundational queries in SQL.

To retrieve all data from Person we would do SELECT * FROM Person;. To retrieve all names from Person we would do SELECT name FROM Person;. To retrieve all unique names from Person SELECT DISTINCT name FROM Person;. Retrieve all unique names from Person that are not alphabetically smaller than 'Joey' we would do SELECT DISTINCT name FROM Person WHERE name >= "Joey";.

Note: Here SELECT is the projection is set theory. FROM is in reference to which Table. DISTINCT means retrieve a unique set. WHERE is the condition statement (selection). AND is an additional condition statement. OR is an additional condition statement. OR NOT is an additional condition statement.

CROSS JOIN, and INNER JOIN/JOIN We will be demonstrating the , keyword in SQL.

Retrieve all instructors who teach a course in which they specialise we would do SELECT Instructor.* FROM Instructor, Class WHERE Instructor.specialisation = Class.name;. Alternatively we could do SELECT Instructor.* FROM Instructor CROSS JOIN Class WHERE Instructor.specialisation = Class.name;.

To reduce overhead of the CROSS JOIN we can instead use INNER JOIN. Thus, we would do SELECT Instructor.* FROM Instructor INNER JOIN Class ON Instructor.specialisation = Class.name;.

Note: .* means everything from that specific TABLE. There is a difference between WHERE and ON.

NATURAL JOIN, and INNER JOIN We will be demonstrating the difference between WHERE and ON.

Retrieve the instructor and their corresponding students we would do SELECT * FROM Teaches NATURAL JOIN EnrolledIn;. Alternatively we could do SELECT * FROM Teaches INNER JOIN EnrolledIn ON Teaches.class = EnrolledIn.class AND Teaches.semester = EnrolledIn.semester;.

Note: We do not need to specify WHERE or ON in a NATURAL JOIN as in is inherit. INNER JOIN keeps duplicate columns whereas NATURAL JOIN does not. The WHERE operator is done after selecting all the tuples. The ON clause happens before JOIN. The WHERE clause happens after JOIN.

OUTER JOIN We will be demonstrating the the OUTER JOIN and the impact of WHERE and ON.

Retrieve all Persons with information of being a student or not we would do SELECT * FROM Person LEFT OUTER JOIN Student ON Person.v_number = Student.v_number;. We can also do RIGHT OUTER JOIN by switching the order; SELECT * FROM Student RIGHT OUTER JOIN Person ON Person.v_number = Student.v_number;.

Note: WHERE and ON do not matter in INNER JOIN. But it does matter for OUTER JOIN. It will return NULL results if there is no match on the right TABLE. LEFT JOIN is the same as LEFT OUTER INNER JOIN.

3.3 Sub-Queries in SQL

Lesson Scope

This lesson covers §6.3 of the textbook. It will cover SQL syntax for writing complex queries that involve sub-queries. Because of the long Reading Break, we will also spend some time reviewing the previous lesson.

Lesson Objectives

1. Given a database schema and a query with a complex sub-goal expressed in plain English or Datalog, you can formulate a correct SQL query with the same semantics.
2. Given a complex SQL query that includes a sub-query, you can describe in plain English the effect of the query.
3. Given a complex SQL query that includes a sub-query, you can rewrite the query so that it does not use a sub-query, explain why that is not possible, or explain why that is not desirable.

Delivery Method

We will visit together a number of sub-queries in SQL.

Preparation

You are encouraged to read the sections of the textbook in advance, but this is not required to follow the lecture. You should be comfortable with converting a complex plain English statement into a declarative logic statement (such as in Datalog) with sub-goals so that we can focus today on syntax.

Lesson Materials

SQL codes written during class.

3.3.1 Sub-Queries Union

We will be using the following DB schema:

Person(<u>v_number</u>, name)
Student(<u>v_number</u>, major)
EnrolledIn(<u>student</u>, class, semester)
Instructor(<u>v_number</u>, specialisation)
Class(<u>code</u>, name)
Teaches(<u>instructor</u>, class, semester)

Sub-Query We will demonstrate how to go about determining the "which" statements for sub-queries.

Retrieve all pairs of different students who are in the same major. Here we will want to know which table(s) and do we need to join any tables. We will also want to know which columns for projection (SELECT) and what conditions for selection (WHERE).

```
SELECT s1.*, s2.v_number
FROM student AS s1 JOIN student AS s2 ON s1.major = s2.major
WHERE s1.v_number < s2.v_number;
```

Note: The AS keyword acts like the rename operator in relational algebra. We use the $<$ symbol instead of the $! =$ symbol to avoid duplicates.

Example difference between $<$ and $! =$ We will use the query from above for retrieving all pairs of different students who are in the same major.

Using $<$ we will return a Table 39...

v_number	major	v_number
333444	CSC	222333
222333	CSC	333444

Table 39: Table Resulting From Than Symbol

And using $! =$ we will return a Table 40...

v_number	major	v_number
222333	CSC	333444

Table 40: Table Resulting From Not Equal Symbol

Retrieve all pairs of classes taught by the same instructor, in descending order of the instructors' v_number.

```
SELECT T1.class, T1.instructor, T2.class
FROM Teaches AS T1 JOIN Teaches AS T2
```

```
    ON T1.instructor = T2.instructor
WHERE T1.class < T2.class
ORDER BY T1.instructor DESC;
```

Note: If we don't specify the ORDER BY it will default to ASC.

Retrieve every student not enrolled in CSC 370. The first part we want to get students enrolled in CSC 370 and then all students not in the first part. We will use a keyword IN.

```
--> Part One
SELECT Student FROM EnrolledIn WHERE class = "CSC 370";

--> Part 2
SELECT v_number FROM Student WHERE
v_number NOT IN (
    SELECT Student FROM EnrolledIn WHERE class = "CSC 370"
);
```

Retrieve all CSC classes that Carol has never taught. First we will retrieve classes that Carol has taught and then we will get CSC classes that are not in the first part.

```
--> Part One
SELECT Class FROM Person JOIN Teaches
    ON instructor = v_number
    WHERE name = "Carol";

--> Part 2
SELECT code FROM Class
WHERE code NOT IN (
    SELECT Class FROM Person JOIN Teaches
    ON instructor = v_number
    WHERE name = "Carol"
)
AND code LIKE "CSC%";

--> Using Correlated Sub-Queries
SELECT code FROM Class WHERE NOT EXISTS (
    SELECT Class
    FROM Person JOIN Teaches ON v_number = instructor
    WHERE name = "Carol" AND class = code
)
AND code LIKE "CSC%";
```

Note: The IN operator is a combination of the OR operator. For example WHERE v_number != 123 OR v_number !=456 is equivalent to WHERE v_number NOT IN (123, 456). In addition, the OR operator cannot be used in a sub-query. LIKE is a string comparison.

Note: Why is using correlated sub-queries bad? Well if we run the sub-query it will not work and it will give us an error. Since we do not know where the class query comes from in the sub-query. However, since we get it with FROM it works as intended. EXISTS is a boolean value. We also do not recommend using correlated sub-queries because the sub-query needs to be run every time as opposed to once on the initial query.

UNION, and UNION ALL We will demonstrate the UNION, and UNION ALL operators.

Find the people who are neither students nor instructors. We can run the following...

```
--> We could do this ...
SELECT * FROM Person WHERE v_number NOT IN (
    SELECT v_number FROM Student
)
AND v_number NOT IN (
    SELECT v_number FROM instructor
);

--> Or we could do this ...
SELECT * FROM Person WHERE v_number NOT IN (
    SELECT v_number FROM Student
UNION
    SELECT v_number FROM instructor
);

--> Will return duplicates if there are duplicates ...
SELECT * FROM Person WHERE v_number NOT IN (
    SELECT v_number FROM Student
UNION ALL
    SELECT v_number FROM instructor
);
```

Note: UNION is a set operator so it does not return duplicates. UNION ALL will return duplicates.

3.4 Grouping, Aggregation, and Sorting

Lesson Scope

This lesson covers §6.4 of the textbook. It will cover SQL syntax for writing complex queries that involve grouping and aggregation. We will focus significantly on the notion of equivalence classes, when attributes needs to be aggregated, and the difference between selection predicates in WHERE versus HAVING clauses.

Lesson Objectives

1. Given a database schema and a query with an aggregation or grouping-based goal expressed in plain English or Datalog, you can formulate a correct SQL query with the same semantics.
2. Given a complex MySQL query that includes a GROUP BY, ORDER BY, HAVING and/or LIMIT clause, you can describe in plain English the effect of the query.
3. When asked, you can state very precisely in less than twenty seconds the difference between a HAVING and a WHERE clause.

Delivery Method

We will walk through queries related to grouping, aggregation, and sorting in SQL.

Preparation

You are encouraged to read the sections of the textbook in advance, but this is not required to follow the lecture.

Optionally, you may wish to review the Math 122 concept of equivalence relations and equivalence classes, such as in this three minute video. If you can relate the GROUP BY operator theoretically to the concept of equivalence classes and functional dependencies, you should reach an elevated level of mastery of this lesson's concepts.

Lesson Materials

SQL codes written during class.

Instructions

Assignment Dataset Run the following in your terminal to import the counties db:

```
mysql -u [username] -p -e "CREATE DATABASE counties;"
mysql -u [username] -p counties < counties.sql
```

3.4.1 Aggregation Group Order

We will be using the County dataset. We will focus on County, CountyPopulation, and ElectionResult tables.

To limit the TABLE to the first two rows we can do SELECT * FROM Country LIMIT 2;. To describe the TABLE we can do DESCRIBE County; which will tell us the field, the type, if it can be null, the key, the default, and extra information. MUL means foreign key.

To show how the table was created we can use SHOW CREATE TABLE County;.

Simple (bag) Projection A projection query to retrieve the population information for each year.

```
SELECT year, population FROM CountyPopulation;
```

Set Projection Query To set a projection query we can use the DISTINCT keyword.

```
SELECT DISTINCT year FROM CountyPopulation;
```

Group By Query A GROUP BY query with same effect with the equivalent classes.

```
SELECT year FROM CountyPopulation GROUP BY year;
```

Aggregation An operation to convert a group of numbers into a single number. Such as COUNT(), SUM(), AVG(), MAX(), MIN(), etc..

```
SELECT year, COUNT(*) FROM CountyPopulation GROUP BY year;
```

To SUM the population for each year we would do...

```
SELECT year, SUM(population)
FROM CountyPopulation GROUP BY year;
```

Note: The condition for an attribute to be successfully projected when there are equivalent classes is that it must be functionally dependent on the field specified in GROUP BY. You can bypass this by including the field in the GROUP BY statement.

Retrieve the total population for each state we can do the following...

```
SELECT year, state, SUM(population)
FROM CountyPopulation JOIN Country ON (county = fips)
GROUP BY year, state;
```

To instead use the attribute 'abbr' in the table 'state' we could do the following...

```
SELECT year, abbr, SUM(population)
FROM CountyPopulation JOIN Country ON (county = fips)
    JOIN state ON (id = state)
GROUP BY year, state;
```

Retrieve only the tuples with year more than 2015 we would do the following...

```
SELECT year, state, SUM(population)
FROM CountyPopulation JOIN Country ON (county = fips)
WHERE year > 2015
GROUP BY year, state;
```

Note: GROUP BY needs to be last in this SQL statement.

HAVING Query The HAVING query filters for the equivalent classes.

Retrieve only the equivalent classes with population sum more than 1000000.

```
SELECT year, state, SUM(population)
FROM CountyPopulation JOIN Country ON (county = fips)
WHERE year > 2015
GROUP BY year, state
HAVING SUM(population) > 1000000;
```

Note: WHERE filters the tuples in selection operation.

We now want to know who won the votes each year. We need to involve the table ElectionResult to check out election information. We can do the following...

```
SELECT SUM(dem), SUM(gop), year
FROM ElectionResult
GROUP BY year;
```

Now we want to retrieve the number of counties won by each party (by grouping on a computed field using SIGN()). We will do the following...

```
SELECT year, SIGN(dem - gop), COUNT(*)
FROM ElectionResult
GROUP BY year, SIGN(dem - gop);
```

We can add ORDER BY year to the statement to make it more readable. We can also do the following SELECT year, SIGN(dem - gop) AS "DemsWON", COUNT(*) to rename the column SIGN(dem - gop) to DemsWON and then if the number is negative they lost and if the number is positive they won. We note that using the SIGN() function we get "duplicate" rows, one with the negative and one with the positive.

Note: The SIGN() function checks if the value is positive.

Retrieve the number of states won by each party per year we would do the following...

```
--> Part One: Determine the number of votes per
--> state per year...
SELECT SUM(dem), SUM(gop), year
FROM ElectionResult JOIN County
ON fips = county
JOIN State ON state = id
GROUP BY state, year;

--> Part Two: Determine the number of states in which
--> that number is higher for each party.
SELECT year, SIGN(dem - gop) AS "DemsWON", COUNT(*)
FROM (
    SELECT SUM(dem), SUM(gop), year
    FROM ElectionResult JOIN County
    ON fips = county
    JOIN State ON state = id
    GROUP BY state, year
 ) AS S GROUP BY year, SIGN(dem - gop);
```

Part two fails because when subquery is used to derive a table for FROM operator (as compared to used in for IN operator), subquery needs to have a name / alias for itself, as if it is a table.

```
--> Part Two: Determine the number of states in which
--> that number is higher for each party.
SELECT year, SIGN(SUM(dem) - SUM(gop)) AS "DemsWON", COUNT(*)
FROM (
    SELECT SUM(dem), SUM(gop), year
    FROM ElectionResult JOIN County
    ON fips = county
    JOIN State ON state = id
    GROUP BY state, year
 ) AS S GROUP BY year, SIGN(SUM(dem) - SUM(gop));
```

This still fails because the returned result from the subquery uses "SUM(dem)" and "SUM(gop)" as the column names, so in the outer query we have to make these as strings instead of being interpreted as aggregation functions. Thus, we need to do the following to use the renaming operator to rename the columns, which is better for readability...

```
--> Part Two: Determine the number of states in which
--> that number is higher for each party.
SELECT year, SIGN('SUM(dem)' - 'SUM(gop)')
AS "DemsWON", COUNT(*)
FROM (
        SELECT SUM(dem), SUM(gop), year
        FROM ElectionResult JOIN County
        ON fips = county
        JOIN State ON state = id
        GROUP BY state, year, SIGN('SUM(dem)' - 'SUM(gop)')
    ) AS S GROUP BY year, SIGN('SUM(dem)' - 'SUM(gop)'));
```

Note: The quoation marks means it is being interpreted as a string.

3.5 NULL's, Table Modifications, and Referential Integrity

Lesson Scope

This lesson covers §6.1.6-6.1.7, §6.5, and §7.1 of the textbook. It will cover the three-valued logic introduced by a third logic value, NULL, and how that influences conditional query operators like joins and outer joins. Additionally, we will look at data manipulation (DML) queries in SQL and how they are affected by referential integrity constraints (i.e., foreign keys).

Lesson Objectives

1. Given a three-valued logic statement, you can correctly identify the result.
2. Given a SQL query that includes joins on tables that contain NULL values, you can correctly identify the result.
3. Given a SQL query that involves an outer join, you can correctly identify the result.
4. Given a referential integrity policy and a SQL DML query that affects a foreign key, you can correctly identify the result.

Delivery Method

In this lesson, we will work together through a series of simple queries on a toy database and try to anticipate the result of the query.

Preparation

You are encouraged to read the sections of the textbook in advance, but this is not required to follow the lecture.

Understanding NULL values properly requires generalising beyond classical boolean logic to a three-valued logic that you have probably not encountered before. This excellent twenty-minute video provides a more comprehensive treatment of the matter than I will provide and is strongly encouraged supplemental material.

Lesson Materials

SQL codes written during class.

3.5.1 Modifications and NULL's

We will start by creating the database; CREATE DATABASE nulls; and nulls;. Next we want to create tables based on the schema (all attributes are int type):

– R (x, y) with x being pk

```
CREATE TABLE R (x INT, y INT, PRIMARY KEY(x));
```

– S (y, z) with y being pk

```
CREATE TABLE S (y INT PRIMARY KEY, z INT);
```

Inserting To insert data into the table we use INSERT.

– (0, 0), (1, 1), (2, 2) -> R

```
INSERT INTO R VALUES (0, 0), (1, 1), (2, 2);
```

– (3, 0) -> S

```
INSERT INTO S VALUES (3, 0);
```

We can also insert from another table.

```
INSERT INTO 'S' (SELECT * FROM 'R');
```

Updating To update we can use the keyword UPDATE.

To update the tuple(s) with x value of 1, to have the y value be 0.

```
UPDATE R SET y = 0 WHERE x = 1;
```

When updating pk attribute(s) we notice that it changes the ordering of the rows. For example, if we update x to be a higher value, the row will now appear lower in the table.

```
UPDATE R SET x = 4 WHERE x = 1;
```

We can also update with sub queries. For the tuple(s) with y value of the ones in S, update y value to 4. We can do the following...

```
UPDATE R SET y = 4 WHERE y IN (SELECT y FROM S);
```

Deleting To delete we use DELETE.

To delete the tuple(s) with x value of 4 we can do the following...

```
DELETE FROM R WHERE x = 4;
```

NULL Values We can now look into NULL values in SQL.

To insert (1, NULL) -> R we can do the following two statements...

```
INSERT INTO R (x) VALUES (1);
INSERT INTO R VALUES (1, NULL);
```

If we want to do comparison on NULL values (update, delete) or do logical operators on NULL values we need to use the IS keyword. Thus, checking for NULLS.

```
UPDATE R SET x = 4 WHERE y IS NULL;
```

Note: NULL means UNKNOWN for our database design. NULLs could happen in practice.

Truth Tables in SQL We can have TRUE, FALSE, and NULL values in a table.

We can start by creating a table and then inserting the values TRUE, FALSE, and NULL.

```
CREATE TABLE three_values(v BOOL);
INSERT INTO three_values (TRUE), (FALSE), (NULL);
```

To create a truth table we can then to a cross join / cartesian product to give all possible combinations of values.

```
CREATE TABLE truth AS (
    SELECT T0.v AS x, T1.v AS y
    FROM three_values AS T0, three_values AS T1
);

SELECT * FROM truth;

SELECT x, y, x AND y FROM truth;
SELECT x, y, x OR y FROM truth;
```

We can treat the AND operator as a MIN() function and we can treat the OR operator as a MAX() function. We can see the AND operator in Table 41 and the OR operator in Table 42.

x	y	x AND y
NULL	1	NULL
0	1	0
1	1	1
NULL	0	0
0	0	0
1	0	0
NULL	NULL	NULL
0	NULL	0
1	NULL	NULL

Table 41: Truth Table x AND y

x	y	x AND y
NULL	1	1
0	1	1
1	1	1
NULL	0	NULL
0	0	0
1	0	1
NULL	NULL	NULL
0	NULL	NULL
1	NULL	1

Table 42: Truth Table x OR y

NULLS and JOIN Operations We can use NULL values in various JOIN operators.

We can do a INNER, NATURAL JOIN by doing the following statement...

```
--> Will return a two column table with x and y.
SELECT T0.v AS x, T1.v AS y
FROM three_value AS T0, three_vaules AS T1
WHERE T0.v = T1.v;

--> Will return a single column table with v.
SELECT * FROM three_values as T0
NATURAL JOIN three_values as T1;

--> Will return a two column table with v and v.
SELECT * FROM three_values AS T0
INNER JOIN three_values AS T1 ON T0.v = T1.v;
```

Note: We do not need to specify a condition for NATURAL JOIN.

We can then use OUTER JOIN to display more information (i.e., the NULL values being returned in the table. Thus, retaining all the information. The NULL value from the two_values table comes from the mismatch of the two tables.

```
CREATE TABLE two_values(v BOOL);
INSERT INTO two_values (TRUE), (FALSE);

--> Will return a two column table with v and v.
SELECT * FROM three_values
LEFT OUTER JOIN two_values
ON three_values.v = two_values.v;
```

UNION and INTERSECTION NULLs and Set Operations such as UNION and INTERSECTION.

Using UNION and UNION ALL will produce two different tables. UNION will produce Table 43 and UNION ALL with produce Table 44.

```
SELECT * FROM three_values
UNION SELECT * FROM two_values;

SELECT * FROM three_values
UNION ALL SELECT * FROM two_values;
```

v
1
0
NULL

Table 43: UNION

v
1
0
NULL
1
0

Table 44: UNION ALL

MySQL does not have INTERSECTION. However, we can simulate this by using INNER JOIN and DISTINCT keywords.

```
--> Won't return NULL values.
SELECT DISTINCT v FROM three_values AS T0
INNER JOIN three_values AS T1
USING(v);

--> Will return NULL values
SELECT DISTINCT v FROM three_values
WHERE (v) IN (
    SELECT v FROM three_value
) OR v IS NULL;
```

Note: USING() operator specifies the common column.

3.5.2 Referential Integrity

We will use the following DB Setup for referential integrity. The constraints introduced here are x needs to be unique and non NULL, same for y.

```
CREATE DATABASE referential_integrity;
USE referential_integrity;
```

```
CREATE TABLE R(x INT, y INT, PRIMARY KEY (x));
CREATE TABLE S(y INT, z INT, PRIMARY KEY (y));
```

When inserting in the presence of constraints related to pk INSERT INTO 'R' VALUES (0, 0), (0, 1); will fail because x needs to be unique.

Foreign Key To add a foreign key we use ALTER TABLE.

```
ALTER TABLE R ADD FOREIGN KEY (y) REFERENCES S(y);
```

Note: You need to have data in your table to create a foreign key in an ALTER statement. Consequently, we cannot do the following DELETE FROM 'S' WHERE 'y' = 0; since it violates the foreign key constraint by removing the reference.

SET NULL Policy and CASCADE Policy Two policies to bypass the foreign key constraints.

To allow for deletion we can have a policy that sets the value of y in R to be NULL if the reference is deleted.

```
ALTER TABLE R ADD FOREIGN KEY (y) REFERENCES S(y)
ON DELETE SET NULL;
```

Note: Here if we ran both ALTER TABLE statements it would be two different constraints being created, thus, DELETE FROM 'S' WHERE 'y' = 0; would still fail because the first ALTER TABLE is still being applied.

We can remove a constraint by doing the following statement...

```
ALTER TABLE R DROP CONSTRAINT r_ibfk_1;
```

Note: *r_ibfk*_1 was randomly generated by MySQL and we can see the constraint name by doing SHOW CREATE TABLE R;.

Instead we can use the cascade policy...

```
ALTER TABLE R DROP CONSTRAINT r_ibfk_2;

ALTER TABLE R ADD FOREIGN KEY (y) REFERENCES S(y)
ON DELETE CASCADE;
```

The policy will then delete both the tuple in S and the tuple in R with the foreign key.

```
INSERT INTO 'R' VALUES (1, 1);
INSERT INTO 'S' VALUES (1, 1);

DELETE FROM 'S' WHERE 'y' = 1;
```

We can also use the UPDATE keyword instead of the DELETE keyword. And the two policies work the same.

Note: Constraints can become messed up if the tables already violate the constraints being imposed.

Unique Keys There is a difference between unique and primary keys.

The difference is unique value can be a NULL value whereas primary keys cannot be NULL.

Note: You can have multiple NULLs for UNIQUE KEY.

```
ALTER TABLE R UNIQUE KEY(y);

INSERT INTO R VALUES (4, NULL), (5, NULL);

SELECT * FROM R;
```

The above should work in MySQL, but may not work in other SQL database systems.

3.6 Check Constraints and Triggers

Lesson Scope

This lesson covers §7.2-7.5 of the textbook. It will cover the creation of more general constraints, such as those that we expressed in relational algebra at the beginning of the course. It will also cover triggers, which, like the CASCADE and SET NULL referential integrity policies, cause updates to a table to “trigger” another change to occur as well.

Lesson Objectives

1. Given a constraint on an attribute expressed in plain English, you can write a SQL query to alter a table to add that constraint.
2. Given the name of a constraint (including key constraints), you can write a SQL query to remove that constraint.
3. Given a constraint expressed in relational algebra, you can write a SQL query to add it as a check constraint to a database.
4. Given a trigger written in SQL, you can correctly explain when it takes and effect and how it affects attributes, tuples, or tables.

Delivery Method

We will walk through queries related to constraints and triggers in SQL.

Preparation

You are encouraged to read the sections of the textbook in advance, but this is not required to follow the lecture.

You may want to review the worksheet on constraints in relational algebra, as this will not be reviewed at the beginning of the lesson.

Lesson Materials

SQL codes written during class.

3.6.1 Constraints

We will be using the following database setup.

```
CREATE DATABASE constraints;
USE constraints;

CREATE TABLE adult(id INT, age INT, PRIMARY KEY (id));
DESCRIBE adult;
```

We have seen two common constraints so far, UNIQUE constraint: try inserting duplicate values and NOT NULL constraint: try inserting NULL value directly and indirectly.

We can add NOT NULL constraint to the age attribute by doing the following...

```
ALTER TABLE adult MODIFY age INT NOT NULL;
```

We can set the default value for age to be 19 by using the DEFAULT constraint.

```
ALTER TABLE adult MODIFY age INT NOT NULL DEFAULT 19;
```

Note: The MODIFY keyword is used to modify attributes (e.g., change data type, add constraint, etc.).

Add attribute CHECK constraint: values for age must be more than 18.

```
ALTER TABLE adult MODIFY age INT
    NOT NULL DEFAULT 19 CHECK (age > 18);
```

Note: We could use both MODIFY and ADD keywords in this case, where the CHECK condition only involves the current attribute of interest. The ADD keyword is used to add constraints to the table, not just specific to an attribute.

```
ALTER TABLE adult ADD CHECK(age > 18);
```

You can delete a constraint by doing the following...

```
ALTER TABLE adult DROP CONSTRAINT adult_chk_2;
```

Add tuple CHECK constraint: involving multiple attributes where age has to be more than id.

```
ALTER TABLE adult ADD CHECK(age > id);
```

Note: Only ADD can be used here, since the CHECK condition involved multiple attributes.

Constraints work on INSERT and UPDATE.

3.6.2 Triggers

Triggers are used to execute some codes when a certain event occurs on INSERT, UPDATE, DELETE.

We will try to achieve the similar effects of the above constraints with triggers instead of checks.

We will create our table adult_trigger...

```
CREATE TABLE adult_trigger(
    id INT, age INT, PRIMARY KEY (id)
);
```

Note: You can store values in MySQL. For example SET @age_limit = 19; where you can get the value by doing SELECT @age_limit;. These values are not stored forever and are session based.

We can create a trigger that changes the value of age to the age limit...

```
CREATE TRIGGER trigger_age
BEFORE INSERT ON adult_trigger
FOR EACH ROW
    SET NEW.age = 19;
```

Here we've created a trigger name trigger_age that inserts on adult_trigger a new age of 19 on each new tuple (i.e., replacing the old value). This trigger is performed before every insertion and goes through each row and sets the value of age to 19.

We can see the trigger(s) by doing SHOW TRIGGERS; and we can delete the trigger(s) by doing DROP TRIGGER trigger_age;.

We can create a trigger that changes the value of age to the age limit only if it is below the limit.

Note: We need to first change the delimiter to allow multiple statements in the terminal. Thus, changing it from DELIMITER ; to DELIMITER $$.

```
CREATE TRIGGER trigger_age
BEFORE INSERT ON adult_trigger
FOR EACH ROW
    BEGIN
        IF New.age < @age_limit THEN
            SET NEW.age = @age_limit;
        END IF;
    END $$
```

Now we only change the age if the condition isn't met.

We can also use the keyword AFTER instead of BEFORE to modify content after insertion. For example, we can create a trigger that after updating adult, it will insert the same old values into adult_trigger.

```
CREATE TRIGGER update_after
AFTER UPDATE ON adult
FOR EACH ROW
    INSERT INTO adult_trigger VALUES (OLD.id, OLD.age);
```

We can now update our adult table and then see an insertion of the old values into adult_trigger.

Note: That if we change AFTER to BEFORE here, we have the same result as the above trigger. When the UPDATE is firstly executed, the system creates a pseudo table called NEW, which contains the new values of the updated tuples. The INSERT operation will then insert the new values from NEW into adult_trigger, which can be done before or after the actual UPDATE changes made.

Example After Trigger

We have the following adult table Table 45...

id	age
21	23

Table 45: Adult

We can then run the following statement to update the table adult for id 21...

```
UPDATE adult SET age = 40 WHERE id = 21;
```

Which will then give us adult table Table 46...

id	age
21	40

Table 46: Adult

And then adult_trigger table Table 47...

Note: NEW is the update values, and OLD is the current values. Also note that there is no OLD row in on INSERT trigger.

id	age
21	23

Table 47: Adult Trigger

3.6.3 Trigger vs Constraint

Trigger can create more constraints and are generally more flexible, because constraints are just already predefined. However, constraints are often more efficient. A trigger is always run before a constraint check.

3.7 ACID Transactions and Concurrency

Lesson Scope

This lesson covers §6.6 of the textbook. It will cover the creation of transactions, i.e., query batches, and the properties that they can facilitate. It will discuss concurrent transactions, particularly those involving database updates, and how various isolation levels provide different definitions of "consistency" in the database.

Lesson Objectives

1. Given a batch of queries, you can write a SQL transaction with an appropriate isolation level.
2. When asked, you can precisely explain the ACID properties and their importance.
3. Given a set of transactions and an isolation level, you can identify all possible database states (both intermediate and final ones).

Delivery Method

We will walk through a slide deck in which we will alternate between new content and collectively brainstorming extensions to that content.

Preparation

You are encouraged to read the sections of the textbook in advance, but this is not required to follow the lecture.

If you have not yet taken CSC 360 and/or you are rusty on the aspects of concurrency covered in that course, it may help to review this seven minute video on race conditions. This knowledge is helpful for understanding isolation levels in this lesson.

3.7.1 SQL Transactions

A transaction is like a batch of queries that allows us to bundle queries together.

It consists of the syntax:

```
START TRANSACTION;
<body>
```

And then either:

COMMIT; to finalise and permanently store all the effects of the query OR
ROLLBACK; to under all the effects of the query.

Note: Concurrent transaction is more than one client viewing or modifying the data at the same time.

3.7.2 ACID Properties

What are we really trying to achieve by bundling queries together as a transaction? We want to support four key database management system (DBMS) properties, known by the acronym: ACID.

- **A**: Atomicity
- **C**: Consistency
- **I**: Isolation
- **D**: Durability

3.7.3 Atomicity

A transaction should occur atomically, i.e., by indivisible. It is never partially committed or roll-backed.

For example, in a bank transfer, we don't want just one query to execute. They must either both occur or neither occur. Otherwise, it breaks the business logic of the application.

Bundling queries together as a transaction is how we specify atomic units of logic, which are otherwise taken to be an individual query.

3.7.4 Consistency

A transaction should not leave a database in an inconsistent state, i.e., violate the data model.

For example, in a bank transfer, we may want the transaction to roll back if an account balance goes negative, if our data model specifies a constraint that accounts must have positive balances.

Our constraints were the best way to ensure consistency. But what if we have a catch-22 where we cannot add a new tuple because of a FK violation, but we cannot add the tuple it should reference either because it has a FK referencing the first tuple? (1-1 relationship).

Note: MySQL checks foreign key constraints immediately; the check is not deferred to transaction commit.

3.7.5 Isolation or "Serialisability"

The results of a transaction should be viewed as independent of other, concurrent transactions.

Consider the following two concurrent transactions:

T1	T2
READ R.y	READ R.z
READ R.x	SET R.x
SET R.x	

Is this safe? What are the possible orders in which these queries could be executed?

3.7.6 Isolation Levels

The isolation guarantees depend on context and affects when locks are acquired. The levels are from least to greatest.

- **ISOLATION LEVEL: READ UNCOMMITTED**: Permits dirty reads, non repeatable reads and phantom reads.
- **ISOLATION LEVEL: READ COMMITTED**: Permits non repeatable reads and phantom reads.
- **ISOLATION LEVEL: REPEATABLE READ**: Permits only phantom reads.
- **ISOLATION LEVEL: SERIALIZABLE**: Does not permit any read errors.

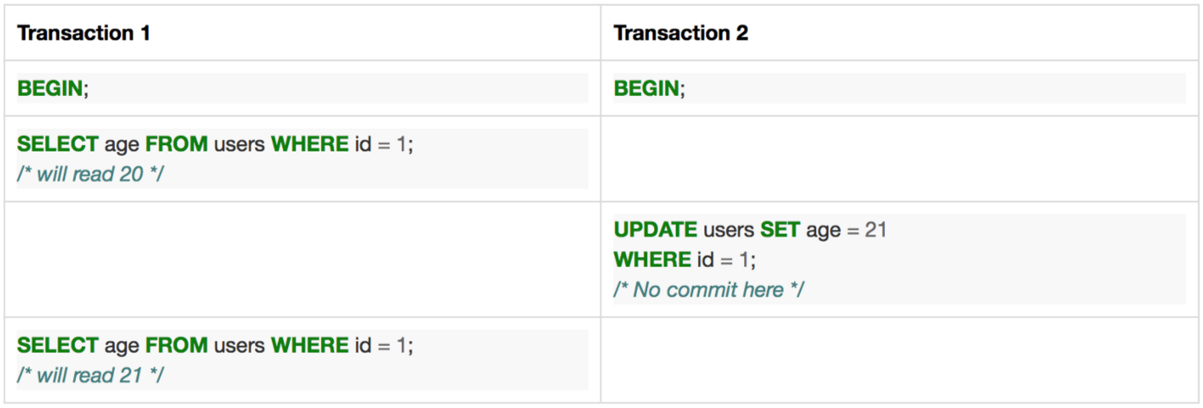
Note: Repeatable Read is the default in MySQL InnoDB and Serializable is the default for SQL in general.

The isolation guarantees depend on the context and affects when locks are acquired.

Example Dirty Reads

A dirty read is when the current transaction reads a row written by another uncommitted transaction that's currently in-flight. Figure 37 illustrates two transactions with dirty reads.

Basically, if the database is not able to keep track of who is changing the data (by keeping multiple versions of the same row with different visibility's) then rows be read even through they should not yet be visible to that other transaction.



Transaction 1	Transaction 2
BEGIN;	BEGIN;
SELECT age FROM users WHERE id = 1; /* will read 20 */	

	UPDATE users SET age = 21
	WHERE id = 1;
	/* No commit here */

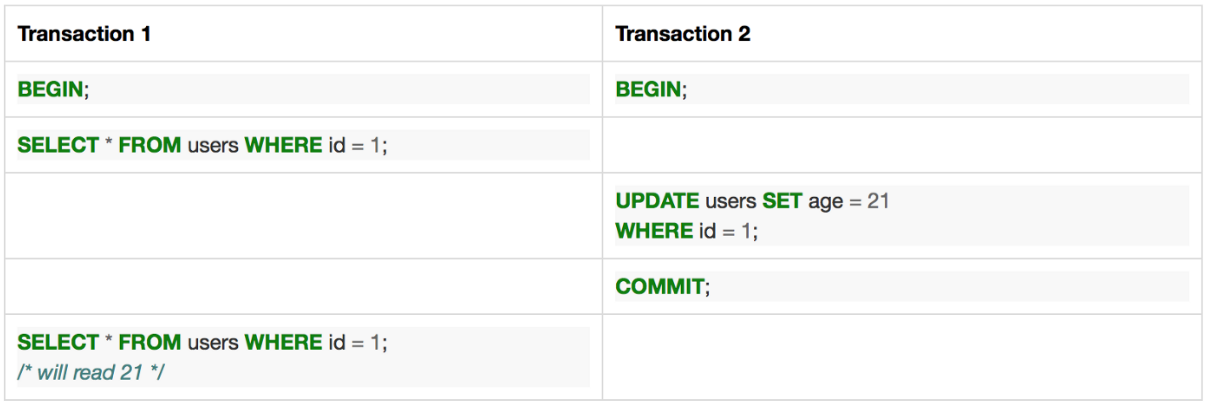
SELECT age FROM users WHERE id = 1; /* will read 21 */	

Figure 37: Dirty Reads [3]

Example Non-Repeatable Reads

A non-repeatable read occurs when the current transaction reads the same data but this time it is different. It is different because another transaction has been committed during the life of the current transaction. Figure 37 illustrates two transactions with non-repeatable reads.

Basically, the database does not maintain what the transaction has already seen so each time the data is read (such as multiple SELECT statements in the same transaction) the same visibility check is done on those rows but some rows may have changed in the mean time.



Transaction 1	Transaction 2
BEGIN;	BEGIN;
SELECT * FROM users WHERE id = 1;	

	UPDATE users SET age = 21
	WHERE id = 1;

	COMMIT;
SELECT * FROM users WHERE id = 1; /* will read 21 */	

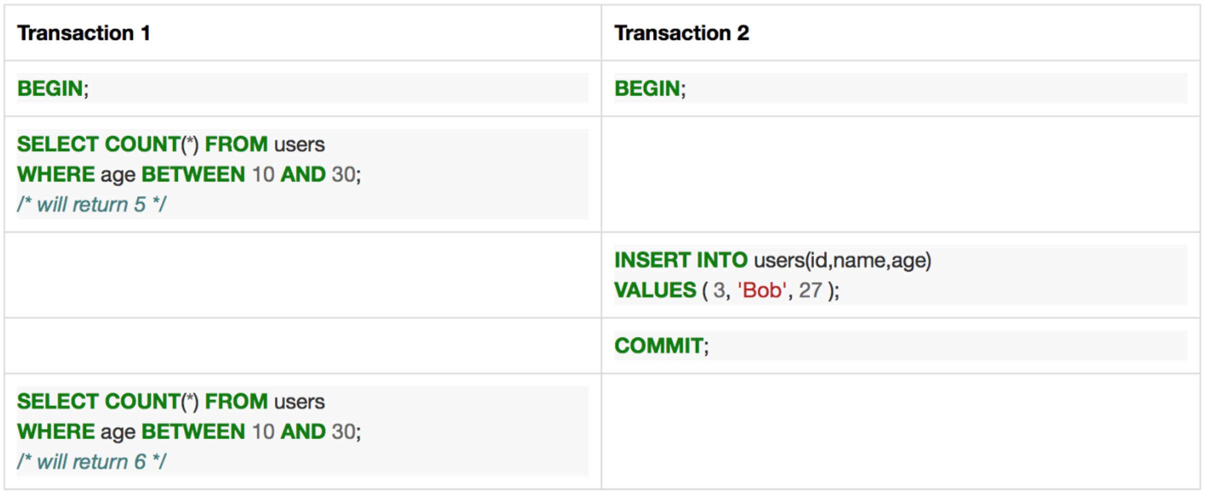
Figure 38: Non-Repeatable Reads [3]

Example Phantom Reads

A phantom read happens when the current transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction. Figure 39 illustrates two transactions with phantom reads.

This can be thought of as a special type of non-repeatable read, but it is important to distinguish it as a different case because the possibility of it occurring depends on the isolation levels explained next.

The distinction is that that the original rows are re-read correctly (even if they had been changed) but a new row (or rows) have been inserted into the range it previously selected. So it hasn't broken the re-read rule, but the data returned is still different.



Transaction 1	Transaction 2
BEGIN;	BEGIN;

SELECT COUNT(*) FROM users
WHERE age BETWEEN 10 AND 30;
/* will return 5 */	

	INSERT INTO users(id,name,age)
	VALUES (3, 'Bob', 27);

	COMMIT;

SELECT COUNT(*) FROM users
WHERE age BETWEEN 10 AND 30;
/* will return 6 */	

Figure 39: Phantom Reads [3]

3.7.7 Durability

Once a transaction has been committed, it is permanent.

For example, in the bank transfer, we don't want somebody to able to revert the transaction later. (It could be possible to issue a second transaction to transfer the money back).

"Permanent" is a strong adjective. How can we ensure the permanence of data? Internal backups, instances of the same data.

3.7.8 Isolation Levels Examples

We will use the following database setup.

```
USE university;
```

Example Dirty Read

We want to read uncommitted so we can do the following...

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

--> T1
START TRANSACTION;
    SELECT * FROM Student WHERE v_number = 222333;
    SELECT SLEEP(5):
    SELECT * FROM Student WHERE v_number = 222333;
COMMIT;

--> T2 (Executed In Another Terminal
-->     Right After T1 Is Run)
START TRANSACTION;
    UPDATE Student SET major = 'MATH'
        WHERE v_number = 222333;
    SELECT SLEEP(6);
    UPDATE Student SET major = 'CSC'
        WHERE v_number = 222333;
COMMIT;

--> Check Value
SELECT * FROM Student WHERE v_number = 222333;
```

Example Committed Read

We want to read committed so we can do the following...

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

--> T1
START TRANSACTION;
    SELECT * FROM Student WHERE v_number = 222333;
    SELECT SLEEP(5):
    SELECT * FROM Student WHERE v_number = 222333;
COMMIT;

--> T2 (Executed In Another Terminal
-->     Right After T1 Is Run)
START TRANSACTION;
    UPDATE Student SET major = 'MATH'
        WHERE v_number = 222333;
COMMIT;
```

```
--> Check Value
SELECT * FROM Student WHERE v_number = 222333;
```

Example Repeatable Read (Phantom Read)

We want to read repeatable so we can do the following...

```
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;

--> T1
START TRANSACTION;
    SELECT * FROM Student WHERE v_number = 444555;
    --> We can use the DO keyword here instead
    --> but the SELECT outputs whereas DO does not.
    SELECT SLEEP(5):
    SELECT * FROM Student WHERE v_number = 444555;
COMMIT;

--> T2 (Executed In Another Terminal
-->     Right After T1 Is Run)
START TRANSACTION;
    INSERT INTO Person VALUES (999999, 'Joey');
COMMIT;
```

Note: That is MySQL, this is prevented at this isolation level! INNODB Transaction Isolation Levels.

Example Serializable

We want to read repeatable so we can do the following...

```
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;

--> T1
START TRANSACTION;
    SELECT * FROM Student WHERE v_number = 222333;
    SELECT SLEEP(5):
    UPDATE Student SET major = 'CSC' WHERE v_number = 222333;
COMMIT;

--> T2 (Executed In Another Terminal
-->     Right After T1 Is Run)
START TRANSACTION;
    SELECT * FROM Student WHERE v_number = 222333;
    SELECT SLEEP(1);
COMMIT;
```

Note: We notice here that T2 does not wait for T1!, because there are no values involved in both transactions being modified / added / deleted. If SERIALIZED, then T2 here should in theory still take much longer in total because it has to wait for T1 to finish first no matter what, as transaction are executed strictly in serial / sequential order.

Final Note: Different user sessions can have different isolation levels.

3.8 Creating Views and Database Indexes

Lesson Scope

This lesson covers Chapter 8 of the textbook. It will cover the creation, purpose, and querying of views and indexes. We will look at their impact on query performance, but also discuss the trade-offs that they present.

Lesson Objectives

1. Given a description thereof in plain English, you can write a correct SQL query to create a view, a materialised view, or an index.
2. Given a view definition and a SQL DML query, you can correctly identify how the base table of the view will be affected, if at all.
3. Given a SQL query, you can identify which database indexes are used by the database management system, if any.

Delivery Method

We will walk through SQL queries related to indexes and views.

Preparation

You are encouraged to read the sections of the textbook in advance, but this is not required to follow the lecture.

Lesson Materials

SQL codes written during class.

3.8.1 Indexes

We will use the IMDB dataset imported from https://datasets.imdbws.com.

```
USE imdb;
SHOW TABLES;

SELECT * FROM name_basics LIMIT 1;
SELECT * FROM title_basics LIMIT 1;
```

Example We will demonstrate slow a query by getting info for actors with name starting with "Robert".

```
SELECT * FROM name_basics
    WHERE primaryName LIKE "Robert%";
```

3.8.2 Introduction Indexes

We would want to use indexes to speed up queries. We can do this by using INDEX.

```
CREATE INDEX 'inx_name' ON name_basics (primaryName);
```

Now if we do the same query...

```
SELECT * FROM name_basics
    WHERE primaryName LIKE "Robert%";
```

Note: We would want to do it on the primary key since doing it on another attribute will not effect the query speed.

To show indexes we can do the following...

```
SHOW INDEX FROM basic_basics;
```

We can also delete an index on a table by doing the following...

```
DROP INDEX inx_name ON name_basics;
```

3.8.3 Indexes and Primary Key

We will use the following database...

```
CREATE database indexes;
USE indexes;

CREATE TABLE data(id INT, x INT, primary key (id));

DESCRIBE data;
```

We will then demonstrate what pk does...

```
INSERT INTO data VALUES (5, 2), (4, 6),
    (1, 3), (2, 5), (3, 4);

SELECT * FROM data;
```

Note: MySQL will already sort by primary key, thus, it will not appear in the order of insertion.

Indexes are created by default when a primary key is created. The primary key acts as an index (primary index). Indexes are also created when creating a unique key, a foreign key, and automatically if none is specified.

Note: The data is stored on the disc at the following locations: */var/lib/mysql* or */opt/homebrew/var/mysql*. You will need to use the command *hexdump − C./file.ibd* to read the files.

We can add some VARCHAR data and check data on the disc.

```
ALTER TABLE data ADD name VARCHAR(16);

UPDATE data SET name = CONCAT('name', id);
```

What will happen if we change the pk value. For example we will change it...

```
UPDATE data SET id = 10 - id;

UPDATE data SET name = CONCAT('newname', id);
```

Next we will use the following database...

```
SHOW DATABASE;
USE information_schema;
SHOW TABLES;
```

We can now check out the table that stores the indexes and triggers.

```
SELECT * FROM INNODB_INDEXES;

SELECT * FROM TRIGGERS;

--> from INNODB_TABLES
SELECT * FROM INNODB_COLUMNS WHERE TABLE_ID = 'someId';
```

Next we can then use indexes...

```
USE indexes;
```

We will now use the keyword EXPLAIN...

```
--> Will return possible_keys as PRIMARY
EXPLAIN SELECT * FROM data WHERE id = 5;

--> Will return possible_keys as NULL
EXPLAIN SELECT * FROM data WHERE x = 5;
```

We can create an index on x which will then return x_index as possible_keys...

```
CREATE INDEX x_index ON data(x);

--> Will return possible_keys as x_index
EXPLAIN SELECT * FROM data WHERE x = 5;
```

Note: There can only be one way of indexing the data. Thus, updating x will not update the ordering of the table since we have the primary index.

3.8.4 Types of Indexes

We have different types of indexes such as cluster index, primary index, and secondary / non-clustered index.

Note: Primary index usually refers to the primary key, but it can be a multi-column index, but it must contain a primary key. If it does not contain a primary key it will be a secondary index.

We can create a multi-column index with the following statement...

```
ALTER TABLE data DROP x_index;

SELECT * FROM WHERE x = 8 ORDER BY name;

CREATE INDEX x_index ON data(x);
CREATE INDEX name_index ON data(name);

--> This one is the best for speeding up the query!
CREATE INDEX multi_index ON data(x, name);

CREATE INDEX multi_index_slow ON data(name, x);
```

The multi_index index is the best because the order matters and we are checking first x and then name.

Note: The order of the multi-column index matters! There is also a may length for indexes. In our case the max length is 3072 bytes.

Why wouldn't we want to index all possible combinations of columns? Well we don't want to create more overhead. It will create more possible_keys options. Another reason is related to the INSERT, UPDATE, and DELETE as it will need to re-indexes everything in our table.

Intuition on Indexes We will consider the following table Table 48 where id is the pk...

id	x
1	2
2	5
2	3
4	6
5	3

Table 48: Indexes Part A

Now if we create an index on x, we would have something like Table 49 instead...

x	id
2	1
3	5
4	3
5	2
6	4

Table 49: Indexes Part B

Now if we do SELECT * FROM data WHERE x = 3;, we would first search for the index to find the id, and then find the actual data in the table.

Note: In module 04 we will learn the data structure behind this, but for now we could think of it as a binary search three.

Here values of x are partitioned into two groups, (2, 3) and (4, 5, 6). So when this query finds x = 3, it first selects group #1, which then divides into (2) and (3), and the later group contains what we want. In this way, we could search in log time, instead of linear time.

3.8.5 Non-Materialized View

Non-materialized views is essentially a virtual table. The data in this virtual table is not stored on disk, but is generated on the fly when the query is executed.

Note: Only materialized views can be used to speed up queries.

We will start with the following database...

```
CREATE DATABASE views;
USE views;

CREATE TABLE data ()id INT PRIMARY KEY, x INT);
INSERT INTO data VALUES (0, 2), (1, 3), (2, 4),
    (3, 5), (4, 6);

CREATE TABLE data_v (x INT PRIMARY KEY, y INT);
INSERT INTO data_v VALUES (2, 2), (3, 3), (4, 4),
    (5, 5), (6, 6);

ALTER TABLE data ADD FOREIGN KEY (x) REFERENCEs data_v(x);
```

We can create a view, for tuples with x value < 6 in data. We can do this with the following statement...

```
SELECT * FROM data where x < 6;

CREATE VIEW x_less_than_6 AS
    SELECT * FROM data WHERE x < 6;

SELECT * FROM x_less_than_6;
```

Note: Views are important for data control. We can create a view with a table that excludes information that we don't want other users to have access to the information (Access Controls).

We can see the view by running...

```
SHOW CREATE TABLE x_less_than_6;
```

Note: The data is not stored on disc at all for non-materialized views (virtual table).

Example Insert, Update, and Delete on Views

```
--> Delete tuple(s) with x = 4
DELETE FROM x_less_than_6 WHERE x = 4;

--> Delete tuple(s) with x = 6;
--> Will get executed even if no tuple(s) exist that
--> satisfies the condition.
DELETE FROM x_less_than_6 WHERE x = 6;

--> Insert tuple (4, 5)
--> The view keeps the referential integrity from the
--> table, so this will fail as it is a dulicate.
```

```
INSERT INTO x_less_than_6 VALUES (4, 5);

--> Insert tuple (10, 1)
--> This will violate the foreign key constraint
--> since we do not have an x = 1 in our
--> data_v table.
INSERT INTO x_less_than_6 VALUES (10, 1);

--> Insert tuple (10, 2)
--> This will work because x = 2 does exist in
--> data_v and it will insert into data.
INSERT INTO x_less_than_6 VALUES (10, 2);

--> Insert tuple (11, 6)
--> This will work, but we will not see
--> the entry in our view, because of the
--> constraint, but we will see it in
--> the data table.
INSERT INTO x_less_than_6 VALUES (11, 6);
```

Complex Queries Views with more complex queries.

We want to create a view for id, data_v.x, y from natural joins between the two tables. We can do this with the following statements...

```
SELECT id, data_v.x, y FROM data
    NATURAL JOIN data_v;

CREATE VIEW view_join AS
    SELECT id, data_v.x, y FROM data
        NATURAL JOIN data_v;
```

Note: The x that we chose matters, because it will have an effect.

We now want to try to delete, also notice how generally DELETE only deletes whole rows). We can do the following...

```
--> This does not work!
--> We would need to specify WHERE condition.
DELETE x FROM data;

--> This does work!
--> All the tuples.
--> DELETE FROM data;
--> DELETE from a JOIN view
--> This will give us the following error...
--> Can not delete from join view views.view_join.
DELETE FROM view_join WHERE id = 100;
```

We can do see insertion and update...

```
--> Insert (100, 1, 2)
--> This will give us the following error...
--> Can not insert into join view views.view_join
--> without fields list.
INSERT INTO view_join VALUES (100, 1, 2);

--> Inserting with the fields list will still
--> result in the following error...
--> Can not modify more than one base table through
--> a join view views.view_join.
INSERT INTO view_join(id, x, y) VALUES (100, 1, 2);

--> Insert (1, 1) for x, y
--> This will suceed since x is from data_v
--> and thus, x and y belong to the same table.
--> This would not work if x is from data (instead of
--> data_v) in the original query for the view!
INSERT INTO view_join(x, y) VALUES (1, 1);

--> Update id from 10 to 101
UPDATE view_join SET id = 10 WHERE id = 5;

--> Update x from 5 to 7
--> This will give a foreign key constraint error!
UPDATE view_join SET x = 7 WHERE x = 5;

--> Change referential integrity policy of fk
--> constraint to CASCADE and try again!
ALTER TABLE data DROP FOERIEN KEY data_ibfk_1;
ALTER TABLE data ADD FOREIGN KEY (x)
    REFERENCES data_v(x) ON UPDATE CASCADE;

--> This will work!
--> This will first update the values in data_v
--> and then in data.
UPDATE view_join SET x = 7 WHERE x = 5;
```

Access Controls We can use views to manage access controls.

We will create a user to test out these access controls.

```
CREATE USER testuser@localhost IDENTIFIED BY '';

GRANT SELECT, INSERT ON table_name TO 'testuser@localhost';

--> Does not work!
DELETE FROM view_join;
```

We will see that the testuser can not DELETE since we didn't grant them permission. Learn more at MySQL Grant.

3.8.6 Materialized View

Materialized view will essentially create another table on the disc.

Note: CREATE MATERIALIZED VIEW statement is not available in MySQL, but we can emulate the statement.

We can create a materialized view for id, data_v.x, y from natural join between two tables.

```
CREATE TABLE view_m AS
    SELECT id, data_v.x, y FROM data
        NATURAL JOIN data_v;
```

Note: If we updated data on view_join it will not be reflected in view_m.

We will try to update the data with the following...

```
UPDATE view_join SET x = 200 WHERE x - 7;
```

This will correctly update view_join, but will correctly not update view_m.

We can update view_m with a TRIGGER, more specifically AFTER TRIGGERS on insertion, update, and deletion...

Note: Indexes and views serve different performance and it depends on the case for which one is better for performance.

3.9 Summary - SQL

3.9.1 SQL

The language SQL is the principal query language for relational database systems. Commercial systems generally vary from the standard.

3.9.2 Select-From-Where Queries

The most common form of SQL query has the form select-from-where. It allows us to take the product of several relations (the FROM clause), apply a condition to the tuples of the result (the WHERE clause), and produce desired components (the SELECT clause).

3.9.3 Subqueries

Select-from-where queries can also be used as subqueries within a WHERE clause or FROM clause of another query. The operators EXISTS, IN, ALL, and ANY may be used to express boolean-valued conditions about the relations that are the result of a subquery in a WHERE clause.

3.9.4 Set Operations on Relations

We can take the union, intersection, or difference of relations by connecting the relations, or connecting queries defining the relations, with the keywords UNION, INTERSECT, and EXCEPT, respectively.

3.9.5 Join Expression

SQL has operators such as NATURAL JOIN that may be applied to relations, either as queries by themselves or to define relations in a FROM clause.

3.9.6 Null Values

SQL provides a special value NULL that appears in components of tuples for which no concrete value is available. The arithmetic and logic of NULL is unusual. Comparison of any value to NULL, even another NULL, gives the truth value UNKNOWN. That truth value, in turn, behaves in boolean-valued expressions as if it were halfway between TRUE and FALSE.

3.9.7 Outerjoins

SQL provides an OUTER JOIN operator that joins relations but also includes in the result dangling tuples from one or both relations; the dangling tuples are padded with NULL's in the resulting relation.

3.9.8 The Bag Model Relations

SQL actually regards relations as bags of tuples, not sets of tuples. We can force elimination of duplicate tuples with the keyword DISTINCT, while keyword ALL allows the result to be a bag in certain circumstances where bags are not the default.

3.9.9 Aggregations

The value appearing in one column of a relation can be summarized (aggregated) by using one of the keywords SUM, AVG (average value), MIN, MAX, and COUNT. Tuples can be partitioned prior to aggregation with the keywords GROUP BY. Certain groups can be eliminated with a clause introduced by the keyword HAVING.

3.9.10 Modification Statements

SQL allows us to change the tuples in a relation. We may INSERT (add new tuples), DELETE (remove tuples), or UPDATE (change some of the existing tuples), by writing SQL statements using one of these keywords.

3.9.11 Transactions

SQL allows the programmer to group SQL statements into transactions, which may be committed or rolled back (aborted). Transactions may be rolled back by the application in order to undo changes, or by the system in order to guarantee atomicity and isolation.

3.9.12 Isolation Levels

SQL defines four isolation levels called, from most stringent to least stringent: "serializable" (the transaction must appear to run either completely before or completely after other transaction), "repeatable-read" (every tuple read in response to a query will reappear if the query is repeated), "read-committed" (only tuples written by transactions that have already committed may be seen by this transaction), and "read-uncommitted" (no constraint on what the transaction may see). Table 50 shows a breakdown of reads.

Isolation Level	Dirty Reads	Non-repeatable Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not Allowed	Allowed	Allowed
Repeatable Read	Not Allowed	Not Allowed	Allowed
Serializable	Not Allowed	Not Allowed	Not Allowed

Table 50: Properties of SQL Isolation Levels

3.10 Summary - Constraint and Triggers

3.10.1 Constraints

Table 51 lists the principal differences among attribute-based checks, tuple-based checks, and assertions.

Type of Constraint	Where Declared	When Activated	Guaranteed to Hold?
Attribute-based CHECK	With attribute	On insertion to relation or attribute update	Not if subqueries
Tuple-based CHECK	Element of relation schema	On insertion to relation or tuple update	Not if subqueries
Assertion	Element of database schema	On any change to any mentioned relation	Yes

Table 51: Comparison of Constraints

3.10.2 Referential-Integrity Constraints

We can declare that a value appearing in some attribute or set of attributes must also appear in the corresponding attribute(s) of some tuple of the same or another relation. To do so, we use a REFERENCES or FOREIGN KEY declaration in the relation schema.

3.10.3 Attribute-Based Check Constraints

We can place a constraint on the value of an attribute by adding the keyword CHECK and the condition to be checked after the declaration of that attribute in its relation schema.

3.10.4 Tuple-Based Check Constraints

We can place a constraint on the tuples of a relation by adding the keyword CHECK and the condition to be checked to the declaration of the relation itself.

3.10.5 Modifying Constraints

A tuple-based check can be added or deleted with an ALTER statement for the appropriate table.

3.10.6 Assertions

We can declare an assertion as an element of a database schema. The declaration gives a condition to be checked. This condition may involve one or more relations of the database schema, and may involve the relation as a while, e.g., with aggregation, as well as conditions about individual tuples.

3.10.7 Invoking the Checks

Assertions are checked whenever there is a change to one of the relation involved. Attribute- and Tuple-Based checks are only checked when the attribute or relation to which they apply changes by insertion or update. Thus, the latter constraints can be violated if the have subqueries.

3.10.8 Triggers

The SQL standard includes triggers that specify certain events (e.g., insertion, deletion, or update to a particular relation) that awaken them. Once awakened, a condition can be check, and if true, a specified sequence of actions (SQL statements such as queries and database modifications) will be executed.

3.11 Summary - Views and Indexes

3.11.1 Virtual Views

A virtual view is a definition of how one relation (the view) may be constructed logically from tables stored in the database or other views. Views may be queried as if they were stored relations. The query processor modifies queries about a view so the query is instead about the base tables that are used to define the view.

3.11.2 Updatable Views

Some virtual views on a single relation are updatable, meaning that we can insert into, delete from, and update the view as if it were a stored table. These operations are translated into equivalent modifications to the base table over which the view is defined.

3.11.3 Instead-Of Triggers

SQL allows a special type of trigger to apply to a virtual view. When a modification to the view is called for, the instead- of trigger turns the modification into operations on base tables that are specified in the trigger.

3.11.4 Indexes

While not part of the SQL standard, commercial SQL systems allow the declaration of indexes on attributes; these indexes speed up certain queries or modifications that involve specification of a value, or range of values, for the indexed attribute(s).

3.11.5 Choosing Indexes

While indexes speed up queries, they slow down database modifications, since the indexes on the modified relation must also be modified. Thus, the choice of indexes is a complex problem, depending on the actual mix of queries and modifications performed on the database.

3.11.6 Automatic Index Selection

Some DBMS's offer tools that choose indexes for a database automatically. They examine the typical queries and modifications performed on the database and evaluate the cost trade-offs for different indexes that might be created.

3.11.7 Materialized Views

Instead of treating a view as a query on base tables, we can use the query as a definition of an additional stored relation, whose value is a function of the values of the base tables.

3.11.8 Maintaining Materialized Views

As the base tables change, we must make the corresponding changes to any materialized view whose value is affected by the change. For many common kinds of materialized views, it is possible to make the changes to the view incrementally, without recomputing the entire view.

3.11.9 Rewriting Queries to Use Materialized Views

The conditions under which a query can be rewritten to use a materialized view are complex. However, if the query optimizer can perform such rewritings, then an automatic design tool can consider the improvement in performance that results from creating materialized views and can select views to materialize, automatically.

3.12 Midterm 03

3.12.1 Scope

This exam covers the SQL unit of the course, including §5–8 of the textbook. The exam assesses your knowledge of SQL and transactions, particularly that:

- given schemata and a plain English or Datalog description of data, you can craft SQL queries to insert, delete, update, or retrieve that data (and vice versa).
- given table instances and a SQL query, you can identify the table instance that results from executing that query against the input tables.
- given schemata and a constraint expressed in relational algebra or plain English, you can implement it as a constraint in SQL (or vice versa).
- given schemata and a plain English description of conditional behaviour, you can write a SQL query to create a trigger that guarantees that behaviour.
- given a description of one or more transactions, you can identify how ACID properties are upheld.
- given an existing SQL query, you can write a new SQL query to construct an index or materialised view that may accelerate the existing query.

3.12.2 Format

This *closed-book* exam is written in-person. If you do not hand in this exam, the weight will be shifted to Assignment 3. The practice exam below is predictive of the style of questions that you can expect. You have up to 45 minutes to write the exam.

4 Query Evaluation and Storage

Overview

Up to this point in the course, we have focused on the design and use of databases, but not really on the management system (i.e., database system internals) that transforms declarative transactions into efficient algorithms that can be executed atomically, consistently, and with isolation guarantees in the presence of other concurrent users.

Moreover, we have seen that durability should be maintained, but not how. In principle, a database should be robust even to disk crashes, fires, earthquakes, and silent bit corruptions.

In this module, we will cover at a high level the implementation of a relational database management system, particularly how it ensures ACID properties while also mapping SQL queries onto data structures and algorithms that can access and update data.

Module-Level Intended Learning Outcomes

Upon completion of this short introductory module, you can:

- given a disk-based algorithm, analyse its complexity with respect to the I/O computational model.
- given a series of SQL DML queries, describe precisely how they change the layout of standard database indexes on affected tables.
- given a recovery method and the contents of disks and log files, restore a database to a consistent state.
- given a SQL DQL query, construct an optimised physical query plan and compare it to the plan selected by a commercial or open source RDBMS.

References

Garcia-Molina, Ullman, and Widom (2009) [2]. Database Systems: The Complete Book. Pearson Prentice Hall. Chapters 13–17.

4.1 I/O Model of Computation

Lesson Scope

As we learned when studying ACID Transactions and Concurrency, relational databases management systems are generally ACID-compliant and therefore guarantee the durability of data. Such a guarantee requires persistent storage; therefore, the physical layer of database design generally involves interacting with (non-volatile) disks. Given the very high latency of reading data from disks, this has significant implications for algorithm design.

Algorithms are always analysed with respect to a model of computation. By default, Computer Scientists generally assume that data is in main memory and apply the Random Access Machine (RAM) model of computation, as is taught in an introductory course to Algorithms and Data Structures, such as in the text by Goodrich and Tamassia. However, there are a wide range of other models of computation, such as the Turing Machine, Parallel Random Access Machine (PRAM), Cell-probe Model, and Automata. In this lesson, we learn the Input/Output (I/O) model that is typically used for disk-based algorithms.

This lesson focuses on §13.3.1 of Garcia-Molina et al. (2007), but uses §13.1–13.2.3 as background context. It also briefly reviews Chapter 1 of Goodrich and Tamassia to establish contrast between the RAM and I/O models. Problem set examples are derived from §15 of Garcia-Molina et al. (2007).

Lesson Objectives

1. When asked, you can define the I/O model of computation in terms of its underlying principle and its cost accounting method.
2. Given basic pseudocode and configuration parameters, you can calculate best-case and worst-case bounds for an algorithm in the I/O model of computation.

Delivery Method

We will follow the corresponding slide deck closely, which is designed to teach the I/O model by means of contrast to the already familiar RAM model.

Preparation

You are advised to review asymptotic analysis in the RAM model from a standard Algorithms and Data Structures course, such as CSC 225, though this will be reviewed briefly in class as well.

4.1.1 Physical Layer

The course has been split into modules that correspond to the abstraction layers in a (relational) database system. (i.e., Conceptual Layer: Module 01, Logical Layer: Module 02, Application Layer: Module 03, and Physical Layer: Module 04).

In this final module, we will answer questions like:

- How is declarative SQL query translated into an efficient algorithm?
- How do we compare algorithms in a relational database?
- How does the DBMS guarantee durability if disks can fail?

We will narrow the scope for this module, as Garcia-Molina et al. suggests it should be a second course on its own.

4.1.2 Modeling

An abstraction is a mental process that we use when we select some characteristics and properties of a set of objects and exclude other characteristics that are not relevant. In other words, we apply an abstraction whenever we want to concentrate on properties of a set of objects that we regard as essential and forget about their differences [2].

4.1.3 Modeling Computation

We construct a *Model of Computation* to determine how to analyse computation. It defines a set of abstractions so that we can more easily "count" what an algorithm does (i.e., computation). There are RAM, Turing Machines, Automata, etc. models.

4.1.4 The Random Access Machine (RAM) Model

Goodrich and Tamassia define a (common) set of high-level primitive operations that are largely independent from the programming language used and can be identified also in the pseudocode. Primitive operations include the following:

- Assignment a value to a variable.
- Calling a method.
- performing an arithmetic operation (e.g., x + y).
- Comparing two numbers.
- Indexing into an array.
- Following an object reference.
- Returning from a method.

This model is called the *RAM Model of Computation*: All memory cells can be accessed with one primitive operation and have the same size. All primitive operations have the same cost, up to a constant factor.

For example, we can count how many operations are in an algorithm.

4.1.5 Asymptotic Analysis

Worst Case Analysis: If we had the worst possible (pathological) input instance, what would be the running time?

Best Case Analysis: If we had the best possible (pathological) input instance, what would be the running time?

Average Case Analysis: If we had a probability distribution over input instances, what would be the expected running time?

Let $f(n)$ and $g(n)$ be functions mapping non negative integers to real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for every integer $n \geq n_0$.

This definition is often pronounced as "$f(n)$ is **big-Oh** of $g(n)$" or "$f(n)$ is **order** $g(n)$".

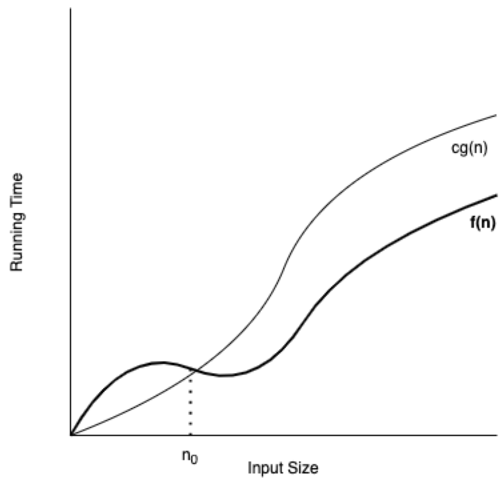


Figure 40: Asymptotic Analysis

The functions in Table 52 are the most commonly observed complexities.

4.1.6 RAM Simplifications

All memory accesses have the same ("unit") cost, i.e., there is no cache hierarchy and no NUMA. All operations are the same cost, i.e., add instructions have the same throughput as an AVX shuffle. There is only one processor. Every machine has the same clock frequency.

Functions Ordered by Growth Rate	Common Name
$\log n$	logarithmic
$\log^2 n$	polylogarithmic
$\sqrt{n}$	square root
n	linear
$n \log n$	linearithmic
n^2	quadratic
n^3	cubic
2^n	exponential

Table 52: Growth Functions

All these exclusions make it easier "to understand, classify, and describe the reality" of computation.

4.1.7 Disk-Oriented Memory Hierarchy

The Drepper hierarchy only showed volatile memory, i.e., memory that is erased if the machine loses power. That is probably not what we want for the data in our relational databases (because it doesn't follow the ACID properties). Garcia-Molina et al. shows a much more detailed hierarchy in Figure 41

Note: most relational databases won't fit in main memory.

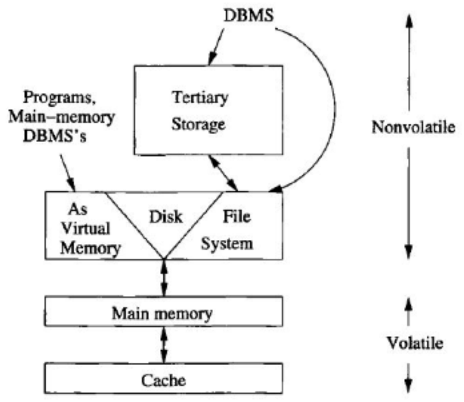


Figure 41: The Memory Hierarchy

4.1.8 Packing Records into Blocks

The same concept applies to tuples. We pack tuples into block aligned to 4KB boundaries. Thus, we can wast space at the end of each tuple and at the end of each block.

header	record 1	record 2	...	record n	///

Table 53: A Typical Block Holding Records

4.1.9 The I/O Computational Model

The time required by a complex algorithm can be described machine-agnostically by:

- Operations in memory are free!
- Each access to a 4KB block of data has unit cost, even if we only need 1 bit.
- The cost of an algorithm is therefore the number of blocks read and written.

Underlying Principle (Dominance of I/O Cost) The time taken to perform a disk access is much larger than the time likely to be used manipulating the data in main memory. Thus, the number of block accesses ("Disk I/Os") is a good approximation to the time needed by the algorithm and should be minimised.

4.1.10 Mechanics of Magnetic Disks

A (magnetic) disk has two components: a disk assembly and a head assembly.

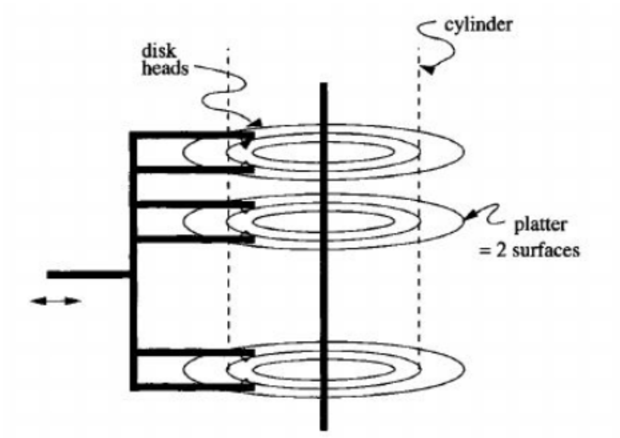


Figure 42: A Typical Disk

Disk Latency = Seek Time + Rotational Latency + Transfer Time.

The Seek Time: up to 10ms, positions the head over correct cylinder.

The Rotational Latency: up to 10ms, rotates disk to start of sector.

The Transfer Time: reads the data contiguously, depends on RPM and amount of data.

We can improve I/O cost by using a sequential layout, using multiple disks, conducting reads out of order with a scheduling algorithm, and prefetching data into memory.

4.1.11 Exclusion of the I/O Model

No consideration, for example, for whether accesses are contiguous or random. We known that for "spinning" magnetic disks, random accesses incur additional seek and rotational latencies.

4.1.12 Summary of I/O Model of Computation

In order to analyse algorithms, we need a Model of Computation, i.e., a set of abstractions. Databases should guarantee durability, so data must be stored on persistent, non-volatile storage. The RAM Model of Computation is agnostic to the block-oriented read pattern of pages from disk and thus inappropriate for disk-bound applications (like external databases). The I/O Model of Computation instead focuses on the number of blocks read/written, based on the Principle of Dominance of I/O (everything that happend in main memory is free).

4.2 B Plus Tree Index Structure

Lesson Scope

As we learned when studying Views and Database Indexes, one of the primary tools to accelerate query performance is to construct indexes. In fact, it is so useful that an index is created automatically in MySQL whenever we designate a primary key so that the database management system can efficiently verify whether new tuples violate the key constraint.

In this lesson, we learn the data structure most commonly used for database indexes, namely the B+-tree, and understand how it facilitates reduced algorithmic cost within the I/O model of computation. While other uni-dimensional indexing strategies exist, the ubiquitous B+-tree is selected by default in MySQL. This lesson focuses on §14.2 of Garcia-Molina et al. (2007); however, it is easier to understand if one is already familiar with RAM-based (2,4)-, (a,b)-, and B-trees as introduced in §20 of Goodrich and Tamassia (2014).

Lesson Objectives

1. Given a series of INSERT, UPDATE, and DELETE SQL queries on a specific database attribute, you can draw a B+-tree with the correct database state.
2. Given a B+-tree and a search key, you can efficiently navigate the tree to establish the existence of the search key and correctly report the corresponding I/O cost.
3. Given a B+-tree, a lower bound search key, and an upper bound search key, you can efficiently navigate the tree to report all tuples matching the range query and correctly report the corresponding I/O cost.

Delivery Method

We will follow the corresponding slide deck closely. It spends a significant amount of time reviewing (2,4)- trees in the RAM model and then shows the adaptations in the B+-tree that make it more suitable for range queries and consistent performance on disk.

Preparation

You are advised to review (2,4)-, (a,b)-, and B-trees in §20 of Goodrich and Tamassia (2014). It may be easier to understand the mechanics of node splitting and upward propagation in the more familiar RAM model before turning to the characteristics of the B+-tree that are specific to I/O accounting.

4.2.1 Simplified Layout of a Table on Disk

Each tuple is stored contiguously as a fixed length record in the order of the attributes in the schema: Column-Oriented and NoSQL databases are out of scope here and for simplicity we ignore varchars.

The records are laid out contiguously in blocks; no record can span across two blocks (i.e., the data is aligned) so some space may be wasted.

In a relation R has 2048 tuples, i.e., T = 2048; each tuple uses 320 bytes; and a block is 4KB, i.e., B = 4096: Then each block contains $\lfloor 4096/320 \rfloor = 12$ records And R spans $\lceil 2048/12 \rceil = 171$ blocks.

4.2.2 Answering Range and Lookup Queries

We recall that SQL is declarative. How is the answer to the following queries determined?

Student($\underline{\text{vnumber}}$, name, age)

Lookup Query: **SELECT** name, age **FROM** Student **WHERE** vnumber = 123456;. Given a search "key", is it in the table instance? If so, return a pointer to it.

Range Query: **SELECT** * **FROM** Student **WHERE** age < 18;. Given a lower and upper bound for attribute x, retrieve pointers to all tuples in the table with an x value between those bounds.

Note: To provide an algorithm, we need to know the data structure!

So, for both of these queries, we would need to do a table scan in which we read in all blocks of the Student relation and check the predicate against each tuple's vnumber or age attribute. This will require $\Theta(T/B)$ I/O's. To do better, we need some sort of search data structure, i.e., an index.

4.2.3 Binary Search Trees (BST's)

What if we used a Binary Search Tree (BST)? A BST stores key-value pairs at nodes; We could store age as a search key and (a pointer to?) the rest of the tuple as a value. A BST has the invariant that at node i with key k_i, every key in the left subtree is less than k_i and every key in the right subtree is not less than k_i.

Now we can use the search tree to more quickly identify the tuples with a matching age value.

Algorithm TreeSearch(k, v):

Input A search key k, and a node v of a binary search tree T.
Output A node w of the subtree $T(v)$ of T rooted at v, such that either w is an internal node

storing key k or w is the external node where an item with key k would belong if it existed.

```
if v is an external node then
    return v
if k = key(v) then
    return v
else if k < key(v) then
    return TreeSearch(k, T.leftChild(v))
else
    return TreeSearch(k, T.rightChild(v))
```

Limitations What are the limitations of this approach? The complexity of BST is $O(n)$ for lookup/insertion/deletion. How do we know which nodes to store together on a block on disk? If we stored node separately, we waste most of each block. A range search could require visiting the entire tree.

To address limitation, we need a balanced tree. There are two main strategies: Use **rotations** to rebalance (e.g., AVL trees, red-black trees, weak AVL trees). Use **node splits** (e.g., 2-3 Trees, 2-4 Trees).

4.2.4 2-4 Trees (An Instance of A-B Trees)

A 2-4 Tree has the following properties:

- **Size Property**: Every node has a fan-out of at least 2 and at most 4.
- **Depth Property**: Every leaf node has the same depth.

This is sufficient to guarantee logarithmic height. The keys in a node are sorted. The i'th subtree contains only keys that are less than the i'th key in the parent and not less than the $(i-1)$'st key in the parent. Therefore, an in-order traversal still retrieves data in sorted order.

Insertion To insert, we need to maintain both properties. This uses a split and grow strategy...

Search for the lead where key k should go and let its parent be v. We add k to v and append a new child to v. This preserves the depth property, because we added a leaf as a sibling of another leaf. But it could violate the size property, i.e., overflow the node and require a split.

To split a node: Replace v with two nodes v' and v'', where v' is a 3-node with children v_1, v_2, v_3 storing keys k_1 and k_2 and v'' is a 2-node with children v_4, v_5 story key k_4. If v was the root of T, create a new root node u; else, let u be the parent of v. Inset key k_3 into u and make v' and v'' children of u, so that if v was child i of u, then v' and v'' become children i and $i+1$ of u, respectively.

Deletion Assume we delete a key from the bottom of the tree. This could cause the node to underflow and require collapsing nodes.

4.2.5 Generalising to A-B Trees and B Trees

With the balanced 2-4 Tree, we addressed the limitation of complexity; our lookup, insertion, and deletion operations are now $\theta(lgT)$. However, we haven't addressed the poor utilisation of blocks.

An A-B Tree has the following properties:

- **Size Property**: Every node has a fan-out of at least a and at most b.
- **Depth Property**: Every leaf node has the same depth.

An order d B Tree has the following properties:

- **Size Property**: Every node has a fan-out of at least $d/2$ and at most d.
- **Depth Property**: Every leaf node has the same depth.

Example Assume a pointer is 8B and a key is 4B. How large can a node be and fit in a 4096B block? Maximise d subject to: $(4d + 8(d+1)) \leq 4096$ (i.e., $d = 340$). So, we could use a 170-340 Tree.

Limitations What are the limitations of this approach? A range search could require visiting the entire tree. Storing tuples in interior nodes decreases fan-out and therefore increases height.

4.2.6 B Plus Trees

The key properties of a B Plus Tree are:

- It is balanced tree with a fixed fan-out, guaranteeing $\log_M(N)$.
- Each node is one block (e.g., 4KB) and therefore one I/O.
- All keys appear sorted in the leaves, which form a linked list.
- Directory nodes have fan-out M and M - 1 keys.
- Subtree of child pointer i only contains values larger then key $i-1$ and strictly less than key i.
- We insert at the leaves and then rebalance by propagating splits up.

Lookup The key properties are subtree of child pointer i only contains values larger than key $i = 1$ and strictly less than key i.

The cost of lookup of key x is the height of the tree.

Range Search The key properties are all keys appear sorted in the leaves, which form a linked list.

The cost of range search of range $[x, y]$ is the height of the tree plus the number of blocks output we need for the search (i.e., 3 + O).

4.2.7 Alternative Search Structures

There are also hash-based indexes (though the B Plus Tree is the default in MySQL).

Advantage Extensible Hashing Lookup, insertion, and deletion with single I/O. Fairly simple to implement.

Advantage of B Plus Tree Control over node occupancy. Support for range queries. Lookup with just a few I/O's. Absolute theoretical guarantees, i.e., no overflow.

4.2.8 Summary of B Plus Trees

Key Properties

- It is a balanced tree with a fixed fan-out, guaranteeing $\log_M(N)$, similar to a B Tree.
- Each node is one block (e.g., 4KB) and therefore one I/O.
- All keys appear sorted in the leaves, which form linked list.
- Directory nodes have fan-out M and M - 1 keys.
- Subtree of child pointer i only contains values larger than key $i - 1$ and strictly less than key i.
- We insert at the leaves and then rebalance by propagating splits up.

Key Advantages

- Guaranteed $\log_M(N)$ I/O's for lookup and $\log_M(N) + O$ I/O's for range query.
- Sorted access to data because leaves are sorted and stored as linked list.
- Insertion works like a B Tree: We insert bottom-up first finding the correct leaf, splitting it if it's already full, and then propagating splits up the tree. This guarantees retain the property of balanced height.

4.3 Multidimensional Indexing and R-Trees

Lesson Scope

As we learned in the previous lesson on B+-Tree Indexes, an index is typically a tree with a very high fan-out in which the height of the tree grows extremely slowly and each node, i.e., input block, contains as much information as possible. Especially if one considers that the top one or two levels of the tree are often cached in memory, this means that a typical lookup, even on a really large table, is only 1–3 I/O's. However, in all the examples we worked on, the key in the B+-tree was a single integer. Indeed, this suggests why we often create an AUTOINCREMENT integer primary key, even when our functional dependencies already suggest the existence of another key.

However, we may have queries that benefit from indexes on other sets of attributes. A classic example is spatial data, where a tuple may be associated with an x- and y-coordinate or a latitude and a longitude. It doesn't often make sense to search for a coordinate based on just one dimension; we are usually interested in both at the same time. A B+-tree does not natively support multi-dimensional search.

In this lesson, we learn the data structure most commonly used for multi-dimensional indexes, namely the R-tree, and understand how it facilitates reduced algorithmic cost within the I/O model of computation. As in the uni-dimensional case, other multi-dimensional indexing strategies exist; however, the ubiquitous R-tree is selected by default in MySQL. Moreover, it is an adaptation to higher dimensions of the B+-tree with which we are already familiar. This lesson focuses on §14.6.7–§14.6.8 of Garcia-Molina et al. (2007).

Lesson Objectives

1. Given a series of INSERT, UPDATE, and DELETE SQL queries on a set of database attributes, you can draw an R-tree with the correct database state and, additionally, indicate how it partitions space.
2. Given an R-tree and a multi-dimensional search point, you can efficiently navigate the tree to establish the existence of the search key and correctly report the corresponding I/O cost.
3. Given an R-tree and an axis-aligned query (hyper-)rectangle you can efficiently navigate the tree to report all tuples matching the range query and correctly report the corresponding I/O cost.

Delivery Method

We will follow the corresponding slide deck closely. It introduces R-trees as an adaptation of B+-trees and focuses on examples of repeated insertion, lookup queries, and range queries.

Preparation

This lesson assumes that you are already comfortable with the material presented in the lesson on B+-Trees.

4.3.1 Selection Operators

We have two algorithms for the selection operator.

Table Scan: Read the table contiguously and check each tuple one-by-one. (i.e., $\Theta(T/B)$).

Index Search: Probe an index for the key and retrieve the corresponding tuple(s) (i.e., $\Theta(lgT/lgB + O)$. The term O will be clarified in a subsequent section.

We can handle predicates on separate attributes with the following options:

- **Option 1**: Table Scan. $\Theta(T/B)$.
- **Option 2**: Use whichever index we have, but what if this isn't ver selective?
- **Option 3**: If we have indexes on both, we could intersect the pointers returned by each index.

4.3.2 Balancing via Node Splits

We want a tree-based index structure to have a few key properties: Balanced height for asymptotic guarantees. High fan-out to maximise block utilisation and decrease I/O's.

We see that the 2-4 and B Trees gives us the mechanism for this: Insert at the leaves and propagate splits up. The tree grows up rather than down, i.e., the tree height changes if and only if we split a tree into two equal-height trees and then connect both newly formed trees to a newly formed root.

4.3.3 R Trees

Definition K "regions"; K pointers. *i*'th pointer gives only regions fully contained within the *i*'th region [inner]. *i*'th pointer gives *i*'th data point; observe greater number of points possible! Occupancy always [(K + 1) / 2, K] pointers. Always balanced, 'cause built up with splits".

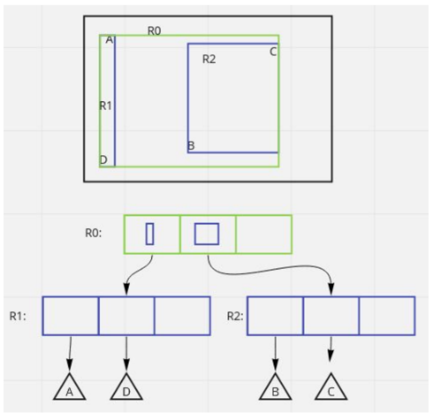


Figure 43: R Tree

Lookup Query We can perform a series of lookup queries.

We have to look at all the rectangles and go down multiple paths.

Range Query We can perform a series of range queries.

We have to look at all the rectangles and go down multiple paths.

Note: R Tree is not heuristic. This is a weakness because there is no theoretical guarantees.

Note: Rectangles can overlap, and we can search multiple paths. We also note that we always create the smallest possible rectangle.

Insertion We start by creating a blank rectangle. When we do an insertion we want to add to the rectangle that will grow the least (i.e., change of area).

4.3.4 Summary of R Trees

An R Tree generalises a B Plus Tree for multiple dimensions, using **containment** instead of order. We construct a minimum bounding box (e.g., multi-dimensional interval) to represent the contents of a sub-tree.

Key Properties

- K "regions"; K pointers.
- *i*'th pointer gives only regions fully contained within the *i*'th region [inner].
- *i*'th pointer gives *i*'th data point; observe greater number of points possible!
- Occupancy always [(K + 1) / 2, K] pointers.
- Always balanced, 'cause built up with splits".

Key Advantages Over B Plus Trees (And Other Alternatives Such As KD Trees)

- All axes are treated equally (greater for spatial data, especially).
- Minimises representation of unoccupied space.
- Points with small Euclidean distance are likely to be stored on the same block.

4.4 Join Algorithms in Secondary Storage

Lesson Scope

In the indexing lessons on B+-Trees and R-Trees, we learned how data structures can be adapted for disk-resident data and how we can redesign main memory data structures for better utilisation of blocks and to minimise I/O's. Except for basic operations on those data structures, we have not really looked at algorithms for disk-resident data.

In this lesson, we learn four distinct algorithms for the join operator. In a database management system (and as we will see in the lesson on query plans), operators are generally independent and data is piped from one operator to the next. If you understand well the algorithms for joins, you can likely understand well the algorithms for the other operators. Moreover, the join is usually the most expensive operator and thus understanding it well gives us good insight into the overall performance of a query. Finally, as you move on past this course, you may encounter techniques for denormalising databases which has a fundamental goal of avoiding joins; so, understanding joins thoroughly will position you well for evaluating the appropriateness of denormalisation. The lesson covers §15.1.3–15.1.4, 15.3.1–15.3.4, 15.5.5, and 15.6.3 of Garcia-Molina et al. (2007).

Lesson Objectives

1. Given a join algorithm, pair of relations, and memory configuration, you can accurately calculate the number of I/O's required to perform the join and correctly state it in asymptotic terms.
2. Given a slow SQL query, you can apply knowledge of join algorithms to know which indexes might accelerate it.

Delivery Method

We will follow the corresponding slide deck closely, which follows one consistent example across four join algorithms (nested loops, block-nested loops, index-based, and hash-based).

Preparation

This lesson assumes familiarity with the JOIN operator and the I/O model of computation. You should be comfortable with both of those concepts in order to follow this lesson effectively.

4.4.1 Calculating Join Results

Assuming the following SQL statements...

```
CREATE TABLE R(x INT, y INT);
CREATE TABLE S(y INT, z INT);
...
SELECT * FROM R NATURAL JOIN S;
```

Algorithmically speaking, how do you compute this join? We have a few options: Nested Loop Join (NLJ), Block Nested Loop Join (BNLJ), Index-Based Join (IBL), and Hash Join (HJ).

4.4.2 Nested Loop Join

Algorithm NestedLoopJoin(R, S)

```
For each r in R
    For each s in S
        If r.join_keys == s.join_keys
            Emit r JOIN s
```

The complexity is quadratic...

$$\lceil T(R) \rceil * \lceil T(S)/2 \rceil$$

4.4.3 Block Nested Loop Join

Algorithm BlockNestedLoopJoin(R, S)

```
For each block br in R
    For each block bs in S
        For each tuple r in br
            For each tuple s in bs
                If r.join_keys == s.join_keys
                    Emit r JOIN s
```

The complexity is...

$$O(B(S) * B(R)/2 + B(R) + B(O))$$

Note: We would want the outer loop to be the smaller relation.

4.4.4 Index-Based Join

Algorithm IndexBasedJoin(R, S, IdxR)

```
For each block bs in S
    For each tuple s in bs
        If s.y in IdxR
            Emit r JOIN s
```

The complexity is...

$$B(S) + 2 * B(S) + B(O)$$

4.4.5 Hash Join

Algorithm HashJoin(R, S)

```
For each block bs in S
    For each tuple s in bs
        Hash s into one bucket
        If bucket is full
            Flush bucket to disk
For each block br in R
    For each tuple r in br
        Hash r into one bucket
        If bucket is full
            Flush bucket to disk
For each bucket b
    BNLJ(b(R), b(S))
```

4.4.6 Summary of Join Algorithms

Nested Loops Join is best if: $B(R) * B(S)$

Block Nested Loops Join is best if: $B(R) * B(S)/2 + B(S)$

Index-Based Join is best if: $B(S) + TreeHeight * T(S) + B(Tree)$

4.5 Logging and Recovery

Lesson Scope

As we learned when studying ACID Transactions and Concurrency, relational databases management systems are generally ACID-compliant and therefore guarantee the durability of data as well as the atomicity of transactions and the consistency of the database state. But what happens if there is, for example, a power outage in the middle of the transaction? How can such properties be truly guaranteed in the presence of uncontrollable external events? In truth, it would be daft to assume such things will never occur, particularly if making claims of ACID-compliance.

In this lesson, we learn the technique of logging in which we write additional metadata to disk in a precise sequence so that if the database suffers downtime, we can confidently return the database to a consistent state in which all transactions have executed (or not!) atomically. The lesson covers §17.1.2–§17.4.3 of Garcia-Molina et al. (2007).

Lesson Objectives

1. When asked, you can correctly report the rules of undo, redo, and undo/redo logging and explain how they relate to The Correctness Principle.
2. Given a log and a checkpointing style, you can accurately identify the conditions required to complete a checkpoint.
3. Given a log and a potentially inconsistent database state on disk, you can apply the log to restore consistency and atomicity.

Delivery Method

We will follow the corresponding slide deck closely, which follows the same examples as the textbook chapter.

Preparation

This lesson expects no specific preparation, though you are encouraged to read the chapter in advance.

4.5.1 Consistency

A database always has a state (i.e., a value for each of its variables). Some states are called *consistent*.

Consistent states satisfy all constraints of the database. They also satisfy *implicit* constraints that the database designer has in mind (e.g., effects of triggers, and logic enacted in the application layer).

4.5.2 Atomicity

There is a clear corollary to the correctness principle: If only part of a transaction executes, the database may be left in an inconsistent state.

The Correctness Principle

If a transaction begins in a consistent state and executes in the absence of any other transactions, then it ends in a consistent state.

As a result, we need to be careful about when we write information to non-volatile storage. For example, if the power goes out, how do we guarantee atomicity. And, what happens if two transactions occur concurrently.

4.5.3 Address Spaces of a Transaction

All operations must occur in registers. Therefore, a typical transaction:

a. Reads data from disk (incurring I/O's) to memory.
b. Loads it from memory to registers.
c. Performs calculations.
d. Stores data from registers back to memory.
e. Writes memory buffers back to disk (incurring I/O's).

Figure 44 shows the log manager and the transaction manager in volatile and non-volatile storage.

4.5.4 Primitive Operations

We have the following important operations that affect state, focused on data movement between the three (key) layers of storage:

INPUT(X): Copies X from disk to a memory buffer.

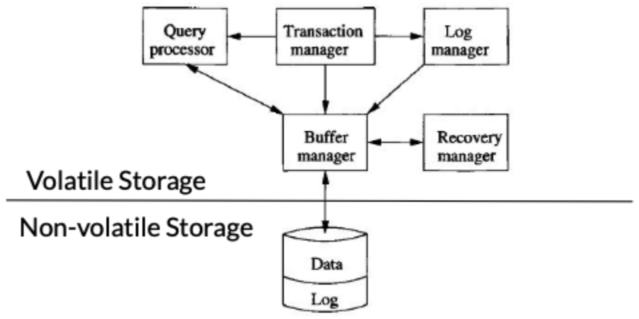


Figure 44: The Log Manager and Transaction Manager

READ(X, t): Transaction reads X from memory into variable t (and invokes INPUT(X) if necessary).

WRITE(X, t): Transaction stores content of variable t into element X in memory (and invokes INPUT(X) if necessary).

OUTPUT(X): Copies X from memory to disk.

The number of calls to INPUT(X) and OUTPUT(X), summer over all X, represent the I/O cost of a database algorithm.

Note: For simplicity assume X is a single value but takes up an entire disk block.

4.5.5 Transaction in Terms of Primitive Operations

Say we have a constraint that $A = 2 * B$ and a transaction that updates doubles the value of A. So we have the following...

Transaction T1:
$A := A * 2$
$B := B * 2$

Here if the constraint is satisfied prior to the transaction the transaction executes in serialisable isolation. Then, the constraint is satisfied after the transaction.

We can write this transaction more explicitly as...

READ(A, t)
$t := t * 2$
WRITE(A, t)
READ(B, t)
$t := t * 2$
WRITE(B, t)

Thus, we can track changes in state as a transaction progresses. We observe that READ implicitly invokes INPUT.

Table 54 shows a detailed transaction of the example.

Here we have a valid state as $A = 2 * B$ thus, consistent.

	T	Memory A	Memory B	Disk A	Disk B
READ(A, t)	42	42		42	84
$t := t * 2$	84	42		42	84
WRITE(A, t)	84	84		42	84
READ(B, t)	84	84	84	42	84
$t := t * 2$	168	84	84	42	84
WRITE(B, t)	168	84	168	42	84
OUTPUT(A)	168	84	168	84	84
OUTPUT(B)	168	84	168	84	168

Table 54: Transaction Example

Example What if the power goes out after OUTPUT(A) and before OUTPUT(B).

As seen in Table 54, it is not true that $A = 2 * B$. As both A and B are 84 in disk.

The transaction was atomic in memory, but the storage was volatile. In non-volatile storage, we did not complete the transfer and the transaction is not atomic.

Example What if the power goes out after WRITE(B, t) and before OUTPUT(A).

Here, we do not violate the constraint of $A = 2 * B$, but what appears in memory is not consistent with what appears in disk.

The transaction was atomic in memory, but that storage was volatile. In non-volatile storage, we did not start the transfer and the transaction is atomic.

Example What if the power goes out after $t := t * 2$.

On disk, there is no evidence that the transaction ever started.

4.5.6 Undo Logging

A log file is an append-only record of important events. If the log file is in non-volatile storage, then we can use to restore a database to a consistent state.

In undo logging, we will use the log to revert (or abort) incomplete transactions. We have the following four operations:

<**START T**>: Transaction T has commenced.

<**COMMIT T**>: Transaction T has completed.

<**ABORT T**>: Transaction T has been reverted.

<**T, X, v**>: Transaction T changed value X, which used to be v.

4.5.7 Undo Logging Rules

Rule U1 If transaction T modifies X, the log record must show <T, X, v> before X is OUTPUT to disk.

Rule U2 If transaction T commits, then its <COMMIT> log record must be OUTPUT to disk after all database elements have already been OUTPUT (but as soon as possible thereafter).

For example, If T modifies X, we get the following sequence of writes to disk:

a. Log records for changes of values.
b. The actual values themselves.
c. The log record that the transaction has committed.

Note: Steps (a) and (b) refer to individual elements, but (c) refers the entire transaction.

4.5.8 Undo Logging Example

We will revisit the previous example, but now adding the logging to the table.

	T	Memory A	Memory B	Disk A	Disk B	Log
				42	84	<START T>
READ(A, t)	42	42		42	84	
$t := t * 2$	84	42		42	84	
WRITE(A, t)	84	84		42	84	<T, A, 42>
READ(B, t)	84	84	84	42	84	
$t := t * 2$	168	84	84	42	84	
WRITE(B, t)	168	84	168	42	84	<T, B, 84>
OUTPUT(A)	168	84	168	84	84	
OUTPUT(B)	168	84	168	84	168	
						<COMMIT T>
FLUSH LOG						

Table 55: Transaction with Logging Example

Example What if the power goes out after OUTPUT(A) and before OUTPUT(B).

Well now in the log file on disk, we can see that this transaction has not completed. Thus, we unwind it and mark that it has been aborted.

Example What if the power goes out after <COMMIT T> and before FLUSH LOG.

Here, we wouldn't need to undo these transactions because we have the COMMIT message for transaction T.

4.5.9 Undo Logging Restoring

To restore we read log backwards and if we see a <T, X, v> record with no previous <COMMIT T> we set X to v, irrespective of its current value.

Example What if the power goes out after OUTPUT(A) and before OUTPUT(B).

	T	Memory A	Memory B	Disk A	Disk B	Log
				42	84	<START T>
READ(A, t)	42	42		42	84	
$t := t * 2$	84	42		42	84	
WRITE(A, t)	84	84		42	84	<T, A, 42>
READ(B, t)	84	84	84	42	84	
$t := t * 2$	168	84	84	42	84	
WRITE(B, t)	168	84	168	42	84	<T, B, 84>
OUTPUT(A)	168	84	168	84	84	
$t := 84$	84			84	84	
WRITE(B, t)	84		84	84	84	<T, B, 84>
$t := 42$	42		84	84	84	
WRITE(A, t)	42	42	84	84	84	<T, A, 42>
OUTPUT(A)	42	42	84	42	84	
OUTPUT(B)	42	42	84	42	84	
						<ABORT T>
FLUSH LOG						

Table 56: Undo Logging Restoring Example

4.5.10 Checkpointing

We don't always want to restore an entire log file. Checkpointing marks points in the log file where restoring up to that point brings us to a consistent state.

For undo logging, to create a checkpoint, we...

- Stop accepting any new transactions.

- Wait until all current transactions end (abort or commit).
- Flush the log to disk.
- Write a log record <CKPT>.
- Flush the log to disk again.
- Resume accepting transactions.

Using rule U2; whenever we reach a <CKPT> while restoring a database, we know all prior transactions have been successfully committed and output to disk.

Example Consider the following log...

```
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
```

Thus, to checkpoint...

```
// we do not allow new transactions
// transactions with T1 and T2
<COMMIT T1>
<COMMIT T2>
<CKPT>
// begin accepting transactions again.
```

4.5.11 Nonquiescent Checkpointing

The trouble with checkpointing as described, however, is that halting all transactions is very disruptive to DBMS throughput.

Instead, we could...

1. Write <START CKPT($T_1, ..., T_k$)>, for all ongoing transactions.
2. Flush the log.
3. Wait for $T_1, ..., T_k$ to complete, but do not prevent other transactions.
4. Write <END CKPT>.
5. Flush the log again.

Example Say we have the following log file on disk at the time of a crash...

```
<START  T1>
<T1 ,  A,  5>
<START  T2>
```

```
<T2, B, 10>
<START CKPT(T1, T2)>
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>
<COMMIT T2>
<END CKPT>
<T3, E, 30>
```

To restore it, start from the end and...

1. Restore E to the value 30.
2. Observe an end checkpoint, so we need not track any as-yet-unseen transactions.
3. Skip <COMMIT T2>.
4. Skip <COMMIT T1>.
5. Skip <T1, D, 20>.
6. Observe <START T3> and note that it has been aborted.
7. End, since all transaction after <END CKPT> have been processed.

4.5.12 Redo Logging

The problem with undo logging is that we constantly have to flush data to disk, costing extra I/O's. In redo logging, we will use the log to reapply (or commit) incomplete transactions. We have the following four operations:

<**START T**>: Transaction T has commenced.

<**COMMIT T**>: Transaction T has completed.

<**ABORT T**>: Transaction T has been reverted.

<**T, X, v**>: Transaction T changed value X to v.

4.5.13 Redo Logging Rules

Rule R1 Before writing element X to disk: All log records <T, X, v> must be output to disk. <COMMIT T> must be flushed to disk.

For example, If T modifies X, we get the following sequence of writes to disk:

a. Log records for changes of values.
b. The log record that the transaction has committed.

c. The actual values themselves.

Note: Steps (b) and (c) have been flipped, relative to undo logging.

4.5.14 Redo Logging Restoring

	T	Memory A	Memory B	Disk A	Disk B	Log
				42	84	<START T>
READ(A, t)	42	42		42	84	
$t := t * 2$	84	42		42	84	
WRITE(A, t)	84	84		42	84	<T, A, 84>
READ(B, t)	84	84	84	42	84	
$t := t * 2$	168	84	84	42	84	
WRITE(B, t)	168	84	168	42	84	<T, B, 168>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	168	84	168	84	84	
OUTPUT(B)	168	84	168	84	168	

Table 57: Redo Logging Restoring Example

To redo, we identify committed transactions. For each <T, X, v>, processing from the start of the log...

a. If T is not committed, do nothing.

b. Otherwise, write v to X.

4.5.15 Undo / Redo Logging

Both undo and redo logging have disadvantages:

- Undo logging requires immediate disk flushes, which could increase I/O's.
- Redo logging requires all data to be kept in memory until the transaction commits, increasing memory footprint.

Undo/Redo logging combines the techniques.

4.5.16 Undo / Redo Logging Rules

Rule UR1 Before writing element X to disk for changes make by transaction T: Create log record <T, X, prior, new> then <COMMIT T> can appear anywhere.

4.5.17 Undo / Redo Logging Restoring

A possible sequence of actions and their log entries using undo/redo logging can be seen in Table 58.

	T	Memory A	Memory B	Disk A	Disk B	Log
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	116	16	16	8	8	<T, B, 8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

Table 58: Undo / Redo Logging Restoring Example

To restore, we go through the log and apply the following steps (necessarily both): **Redo** all the committed transactions in the order earliest-first. **Undo** all the incomplete transactions in the order latest-first.

4.5.18 Summary - Logging

We maintain a log file of data modifications and transactions starts/ends. Like version control does for source code, this records every state the database has been in.

With undo logging, we write after a transaction ends and undo lost transactions.

With redo logging, we write prior to a transaction ending and reapply lost transactions.

Undo / Redo logging creates a larger log file but is more flexible than either technique. We can record a commit whenever we like, but in restoration we need to undo *all* incomplete transactions and redo *all* complete transactions.

4.6 Parity and RAID Schemes

Lesson Scope

As we observed in the previous lesson on Logging and Recovery, ACID-compliance necessitates acknowledging that things can go wrong and therefore taking preventative action so that we can recover from those events. However, an astute learner might observe that this only works if we don't lose the log file as well. What happens if the disk itself is corrupted or destroyed? The half-life (or "mean time to failure") of a set of disks is ten years. Clearly, for a long-lived database, a log file alone is insufficient to guarantee durability.

In this lesson, we learn techniques of redundant storage in which we record extra data or extra copies of data the enable us to restore database state, even in the case of unrecoverable disk loss. The Redundant Array of Independent Disks (RAID) schemes, moreover, provide a cost-effective way to reduce redundancy while also guaranteeing resilience. The lesson covers §13.4 of Garcia-Molina et al. (2007).

Lesson Objectives

1. Given a checksum, you can verify whether or not an intermittent failure has occurred
2. Given a RAID scheme and set of disks that have been subjected to failure, you can restore all disks to their correct state

Delivery Method

We will follow the corresponding slide deck closely, which follows the same examples as the textbook chapter.

Preparation

This lesson assumes familiarity with xor logic. No specific preparation is required, though you are encouraged to read the chapter in advance.

4.6.1 Durability and RAID

In the event of a disk failure, we will be able to detect the failure and recover data using common parity-based schemes such as RAID. We will start by a quick review of mod-2 arithmetic.

Mod-2 Arithmetic (i.e., XOR) Within modulo arithmetic, we have the nice property that...

$$\begin{aligned} &(x+y)(mod\,2) = z \\ \Rightarrow\ &(x+z)(mod\,2) = y \\ \Rightarrow\ &(y+z)(mod\,2) = x \end{aligned}$$

Example Simple Examples.

$$\begin{aligned} (1+1)(mod\,2) &= 0 \\ (0+0)(mod\,2) &= 0 \\ (1+3)(mod\,2) &= 0 \\ (1+1+0+1+0+0+2)(mod\,2) &= 1 \end{aligned}$$

4.6.2 Types of Disk Failures

There are four primary types of disk failures:

- **Intermittent Failure**: A read fails but a subsequent attempt succeeds.
- **Media Decay**: Some bits are corrupted and retries are unsuccessful.
- **Write Failure**: Neither reads nor writes are successful (e.g., due to power loss).
- **Disk Crash**: Whole disk is permanently lost.

The above is in order of severity from least to most.

Note: Maintaining durability requires addressing all three types of failures.

4.6.3 Intermittent Failures

A read fails but a subsequent attempt succeeds. Observe that retrying will be successful. We simply need to detect the failure. We use "checksums" to accomplish the detection.

4.6.4 Checksums

Say we read byte 01001110. How do we know if it was successful? The basic idea is...

Let's append a "parity but" that we can check. Parity bit is set to the mod 2 sum of all bits in the byte. We can think of it as a bool flag indicating if there are an odd number of bits set. This guarantees the parity of the entire string is always even.

Example Consider the following

$$\begin{array}{l} 01001110: \ 0 \\ 01111111: \ 1 \\ 01010101: \ 0 \end{array}$$

What about detection? What if the parity bit is the one that is incorrect? What if more than one bit is incorrect?

Note: It doesn't matter whether we lost actual data or checksums data, the test will still work. This test is not sufficient on its own (a 50% chance: if it's odd we can, but if it's even we can't).

We can thus have two parity bits. If we add k parity bits we can drive this probability of discovery up to $1 - 2^{-k}$. However, how do we add more then one parity bit? We will create two components and add a parity for each component.

Thus we have the following...

$$0100 \quad 1110 \quad 11: \ 0$$

Here $k = 2$, and we can drive this arbitrarily high to get the probability of 2^{-k}. However, it doesn't fully allow us to detect if it might be incorrect. It has just driven the probability down.

Note: Checksums are helpful for identifying the problem, but it doesn't do anything for restoration in the case of permanent data corruption.

4.6.5 Media Decay and Write Failures

Media Decay: some bits are corrupted and retries are unsuccessful.

Write Failure: Neither reads nor writes are successful (e.g., due to power loss).

Say we cannot read the data, even after multiple attempts? Now what?

We need **redundancy** so we can actually restore the value. This is the idea of Stable Storage.

4.6.6 Stable Storage

We pair up each sector with another sector, call them X_L and X_R. We will redundantly write data X to X_L and then X_R (both with checksums).

1. **Write Procedure**:
   (a) Continually try to write X_L until checksum passes.
   (b) Then continually try to write X_R until checksum passes.
   (c) If either fails after k attempts, assume failure and replace that sector.
2. **Read Procedure**:
   (a) Alternate reading X_L or X_R and verify checksum.

Note: We lost half our storage here on a disk with this technique.

4.6.7 Stable Storage Guarantees

Media Failure: If either X_L or X_R fails, we can always use the other to restore data.

Write Failure: If failure occurs writing X_L, then X_R is still "good". If failure occurs writing to X_R, then X_L has already successfully completed.

What if the second one fails while restoring the first one?

4.6.8 Disk Crash

A disk crash is when a whole disk is permanently lost.

How do we extend these ideas to handle complete disk crashes? We need redundancy across disks. These schemes are called RAID (i.e., Redundant Array of Independent Disks).

4.6.9 RAID / Mirroring

The basic idea is that we create one extra copy of every data disk, called a "redundant disk".

Example Odd of failure, assuming 10 year meant time to failure...

4.6.10 RAID 4

Clearly, creating an entire duplicate of the database is unfeasible in many cases. We can use ideas of parity instead. The basic idea is that we create only one extra disk which stores parity of every other disk.

Note: We use XOR here with the corresponding bit in each disk.

Example Consider the following...

```
Disk 0: 11101001
Disk 1: 01010001
Disk 2: 01110111
```

The parity disk would be... 11001111. This will be our backup disk.

And recovery will be... no matter which disk goes down, we can recover it by taking the XOR of all the other disks.

Using the following properties...

$$\begin{aligned} (x \oplus y \oplus z) &= w \\ \Rightarrow (w \oplus x \oplus y) &= z \end{aligned}$$

Thus, if we need to recover disk three we can use disk 0, disk 1, and the parity disk.

Example What if we wanted to update disk 2?

We would update the parity disk by first calculating the XOR of the old and new value of disk two and then XOR the result with the parity disk.

```
Disk 0: 11101001
Disk 1: 01010001 –> 00000000
Disk 2: 01110111
```

Thus, changes will be old $\oplus$ new, which in this case will be 01010001.

Then if we do... 11001111 $\oplus$ 01010001 we get a new parity disk of 10011110.

4.6.11 RAID 5

The problem with that approach is maintenance. Every write has to update the parity disk. The basic idea is that we rotate the parity block across each disk.

```
Disk 0: 11101001
Disk 1: 01010001
Disk 2: 01110111
```

Thus, if there are $n+1$ disks numbered 0 through n, we could treat the ith cylinder of disk j as redundant if j is the remainder when i is divided by $n+1$.

We are then treating each disk as the redundant disk for some of the blocks.

4.6.12 RAID 6

What if we lose multiple disks? We cannot solve that with a parity disk alone. The basic idea is that we use *error-correcting codes*, such as Hamming Code.

Thus, we use some number $2^n - 1$ disks. We then form an identity matrix with the redundant disks. Every binary number > 0 appears as some column.

Note: Disk 5, 6, 7 form an identity matrix of size three. The remaining disks will have every single bit pattern of length three that has at least two one sets. The assignment is arbitrary. We have then created three separate arrays of redundancy.

Here we have that every array is going to have exactly four disks and every one of these disk is going to participate in at least two arrays.

Restoration / Parity Bits: Redundant disk *i* is the parity disk of other 1 disks. If any two disks fail, there must be a parity disk that we can still use to restore it.

Example Consider the following...

$$\begin{array}{r|cccc|ccc}
\text{Disk Number} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
 & 1 & 0 & 1 & 1 & 0 & 0 & 1
\end{array}$$

Write to disk 2 the value: 00001111. Disk 2 parity contribution = 10101010 $\oplus$ 00001111 $=$ 10100101.

Disk 2 in the matrix has bits set in row 1 and 2. Therefore, we update parity disks 5 and 6.

Parity 5 $=$ 01100010 $\oplus$ 10100101 $=$ 11000111
Parity 6 $=$ 00011011 $\oplus$ 10100101 $=$ 10111110

Before

Disk	Contents
1	11110000
2	10101010
3	00111000
4	01000001
5	01100010
6	00011011
7	10001001

After

Disk	Contents
1	11110000
2	00001111
3	00111000
4	01000001
5	11000111
6	10111110
7	10001001

Observe that all parities from the matrix are still correct after the write to disk 2.

Example Consider that disk 2 and 5 are both lost.

We look for a matrix row in which they have different bits set. In row 2, disk 2 has a 1 but disk 5 has a 0. Therefore, disk 2 can be restored from disk 1, 4, and 6. Thereafter, disk 5 can be trivially restored (only one failure).

Thus, we can do the following 11110000 $\oplus$ 01000001 $\oplus$ 10111110 $=$ 00001111.

Before	
Disk	Contents
1	11110000
2	????????
3	00111000
4	01000001
5	????????
6	10111110
7	10001001

After	
Disk	Contents
1	11110000
2	00001111
3	00111000
4	01000001
5	????????
6	10111110
7	10001001

To recover disk 5, we see that it only participates in the first array. Thus, we can restore with disk 1, 2, and 3. We will then obtain 11110000 $\oplus$ 00001111 $\oplus$ 00111000 $=$ 11000111.

Example Consider that disk 1 and 4 are both lost.

We start by looking at column 1 and column 4 and find an array where they have different bits. In this case the first array. We see can that we can recover disk 1 with disk 2, 3, and 5.

Thus, we can do the following 10101010 $\oplus$ 00111000 $\oplus$ 01100010 $=$ 11110000

Before	
Disk	Contents
1	????????
2	10101010
3	00111000
4	????????
5	01100010
6	00011011
7	10001001

After	
Disk	Contents
1	11110000
2	00001111
3	00111000
4	????????
5	11000111
6	10111110
7	10001001

We can then recover disk 4 with any array that it participates in, which in our case is array 2 and 3.

Example Consider that disk 1 and 2 are both lost.

We look for a matrix row in which they have different bits set. In row 3, disk 1 has a 1 but disk 2 has a 0. Therefore, disk 1 can be restored from disk 3, 4, and 7. Thereafter, disk 2 can be trivially restored (only one failure).

Thus, we can do the following $00111000 \oplus 01000001 \oplus 10001001 = 11110000$

Before	
Disk	Contents
1	????????
2	????????
3	00111000
4	01000001
5	11000111
6	10111110
7	10001001

After	
Disk	Contents
1	11110000
2	????????
3	00111000
4	01000001
5	11000111
6	10111110
7	10001001

4.6.13 Summary

There are four types of disk failures. We need to be able to handle all of them: Checksums for error detection, Stable Storage for bad sectors, RAID schemes for disk crashes.

4.7 Logical Query Plans

Lesson Scope

Throughout this course, we have emphasised the strengths of the abstractions in a relational database management system. One particularly influential abstraction is that queries are expressed in a declarative programming language, namely SQL. The physical layer of the database is responsible for parsing the query and transforming it into something procedural, i.e., a query plan. Even this process, however, is split by abstraction in which we first construct a logical query plan that indicates what should be done, before it is translated into a physical query plan that indicates how.

In this lesson, we learn how to construct a logical query plan from a SQL query. Moreover, we learn how to apply algebraic laws to convert logical query plans into other, equivalent logical query plans, thereby creating multiple options for how a query could be executed. The lesson covers §16.1–16.3 of Garcia-Molina et al. (2007).

Lesson Objectives

1. Given a SQL query, you can convert it into some equivalent logical query plan.
2. Given a logical query plan, you can convert it into some equivalent SQL query.
3. Given two logical query plans, you can identify the algebraic laws to transform one plan into the other or, if they are not equivalent, explain why.

Delivery Method

We will follow the corresponding slide deck closely, which itself follows the textbook closely.

Preparation

This lesson assumes familiarity with relational algebra. You should be comfortable converting SQL queries into relational algebra and vice versa.

4.7.1 Extended Relational Algebra

Why more relational algebra? We have seen a few limitations of relational algebra now that we've encountered SQL:

- It doesn't handle bags, but commercial RDBMS's do (Why?).
- It doesn't handle several operators that we have seen, such as group by and sort.
- We will use it for our query optimisation.

Question How do we answer a SQL query?

This is performed by the query execution engine, using the following steps:

- A query compiler parses the tokens to produce a parse tree.
- The parse tree is converted into a relational algebra tree, called a *logical query plan*.
- The logical query plan is optimised using heuristics / rules of thumb.
- The logical query plan is converted into a *physical query plan* by selecting algorithms, assigning buffers, and determining how to transfer results between operators.

Observe that all of these is part of the query response time, so it need to be done quickly!

4.7.2 Projection on Bags

A projection on a bag simply discards columns, it doesn't remove duplicates.

Example Here we project onto A using a bag projection:

A	**B**
1	2
3	4
1	3

A
1
3
1

4.7.3 Union on Bags

A union on a bag simply concatenates two relations, it doesn't remove duplicates.

Example Here we union this relation onto itself using a bag union:

A	**B**
1	2
3	4
1	3

A	**B**
1	2
3	4
1	3
1	2
3	4
1	3

4.7.4 Intersection on Bags

An intersection on a bag has a copy for each paired row.

Example Here we intersection these relations using a bag intersection:

A	B
1	2
1	2
1	3

A	B
1	2
1	2
1	2

A	B
1	2
1	2

Note: The count of each row in the result will be the min of the counts from each operand relation.

4.7.5 Set Difference on Bags

A set difference on a bag will have each row on the right hand side "cancel out" a row on the left hand side.

Example Here we take the set difference of the left relation from the right relation:

A	B
1	2
1	2
1	3

A	B
1	2
1	2
1	2

A	B
1	2

Note: We see that there is one copy of (1, 2) remaining after two are cancelled out by tuples in the LHS.

4.7.6 Selection on Bags

A selection on a bag simply applies a predicate to each row independently.

Example Here we select on $B = 2 * A$ using a bag selection:

A	B
1	2
3	4
1	3
1	2
3	6
1	3

A	B
1	2
1	2
3	6

4.7.7 Cross Product on Bags

A cross product on bags will always produce *nm* result tuples even if there are duplicates.

Example Here we take a cross product of the two relations using a bag cross product:

A	B
1	2
1	2
3	6

C	D
3	5
1	2

A	B	C	D
1	2	3	5
1	2	1	2
1	2	3	5
1	2	1	2
3	6	3	5
3	6	1	2

4.7.8 Duplicate Elimination

Sometimes we want to transform a bag into a set. For that, we use the duplication operator (i.e., δ).

Example Here we apply it to this input relation, R(A, B):

A	B
1	2
1	2
3	6

A	B
1	2
3	6

4.7.9 Extended Projection

We can also project relations into new attributes, e.g., with arithmetic expressions. For that, we extend the projection operator (i.e., π) and use an arrow notation to rename attributes (i.e., $\rightarrow$).

Example To take our LeagueTable relation and add a GamesPlayed attribute, we could do:

$$\pi_{(league, team\_name, wins, losses, draws, wins+losses+draws \rightarrow games\_played)}(LeagueTable)$$

4.7.10 Group By / Aggregation Operator

We can also group tuples of a relation based on common values of an attribute. However, for all other attributes, we need to define an aggregation to apply to them: SUM, MAX, MIN, AVG, or COUNT.

Example If we group the tuples below by A and take the max B, we can apply:

$$\gamma_{(A,MAX(B)\rightarrow B)}(R)$$

A	**B**
1	2
1	5
3	6

A	**B**
1	5
3	6

4.7.11 Sorting Operator

Sets are naturally unordered. The sorting operator (i.e., τ) sorts a relation. Observe that this may have *no effect* if applied prematurely!

Example Here we sort the relation on attribute B:

$$\tau_{(B)}(R)$$

A	**B**
1	2
3	6
1	5

A	**B**
1	2
1	5
3	6

4.7.12 Outer Join Operator

An extension to the join operator is the outer join. Sometimes, it is convenient to know which tuples in one table *did not* have a match with tuples in the other table. These are called dangling tuples.

An outer join performs a regular (inner) join and unions that with each tuple paired with null values (i.e., $\perp$) from the other relation.

Example Here we perform an outer join on $A = A'$:

A	**B**
1	2
3	6
1	5

A'	B'
1	2
2	5

A	**B**	A'	B'
1	2	1	2
3	6	$\perp$	$\perp$
1	5	1	2
$\perp$	$\perp$	2	5

4.7.13 Summary

In order to support the full functionality of a modern RDBMS, it is necessary to extend relational algebra to:

- Support bags / multisets so that duplicate objects can be represented.
- Include additional operators, including:
    - More powerful projection that can add derived columns as well.
    - Sorting.
    - Grouping and aggregation.
    - Duplicate elimination (to restore sets from multisets).
    - Outer joins.

4.7.14 SQL Query Speed

Observe the following three SQL queries...

```
 SELECT R.x
FROM R
     JOIN S ON (R.x = S.x);

 SELECT *
FROM (SELECT x FROM R) AS Rx
     NATURAL JOIN (SELECT x FROM S) AS Sx;

 SELECT R.x
FROM R, S
WHERE R.x = S.x;
```

We now want to order them from fastest to slowest. However, they are all identical! Remember SQL is declarative and these all express the same thing.

So how do we transform a declarative query into something algorithmic?

4.7.15 Basic Algebraic Identities

The Law of Commutativity An operator is *commutative* if its operand can always be permuted without changing the result of the operator (i.e., addition and multiplication).

Note: difference and division are not!

The Law of Associativity An operator is *associative* if repeated applications can be permuted without changing the result of the operator (i.e., addition and multiplication).

4.7.16 Arithmetic Expression Trees

On application of a binary tree is to represent an arithmetic expression:

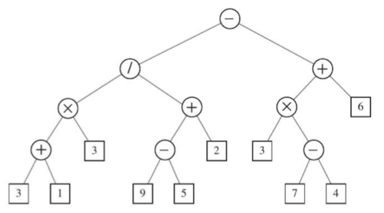


Figure 2.18: A binary tree representing an arithmetic expression. This tree represents the expression $((((3 + 1) \times 3)/((9 - 5) + 2)) - ((3 \times (7 - 4)) + 6))$. The value associated with the internal node labeled "/" is 2.

Figure 45: A Binary Tree Representing An Arithmetic Expression

From the tree in Figure 45, we build a unique arithmetic expression:

1. If there is a left child:
   (a) Print an open parenthesis ’(’.
   (b) Visit the left child.
2. Print the label of the node.
3. If there is a right child:
   (a) Visit the right child.
   (b) Print a close parenthesis ’)’.

Note: The procedure is deterministic, so the expression is unique per tree.

Algorithm Calculate Result

calc_result(node v):

1. If v is a leaf:
   (a) Return its label(v).
2. Else:
   (a) Apply operator given by label to calc_result(left_child) as left operand and calc_result(right_child) as right operand.

Note: The procedure is also deterministic, so every node is associated with a unique value.

4.7.17 Arithmetic Expression Tree Transformations

Consider again our algebraic laws: We can swap the left and right children of a commutative operator.

Therefore, an equivalent arithmetic expression tree can be found in Figure 46 which applies a swap on one addition operator.

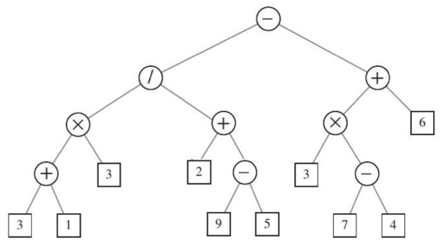


Figure 46: An Alternative Binary Tree Representing An Arithmetic Expression

Consider again our algebraic laws: We can rotate a node and its child if they are both the same associative operator.

There, an equivalent arithmetic expression tree can be found in Figure 47 which applies a rotation on two multiplication operators.

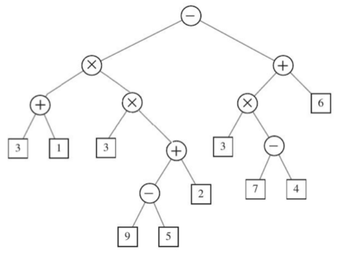


Figure 47: An Alternative Binary Tree Representing An Arithmetic Expression

Example How many trees (i.e., expressions) are equivalent to the one given in Figure 48?

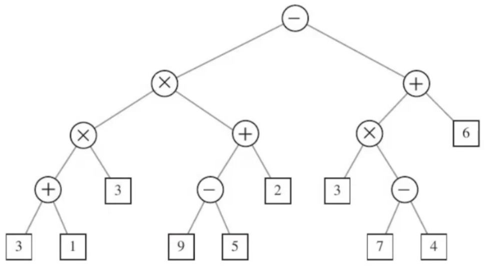


Figure 48: An Alternative Binary Tree Representing An Arithmetic Expression

There are: 3 commutative difference operators, 3 commutative addition operators, 1 associative pair of multiplications.

Therefore, we have $2^{3+3+1} = 2^7 = 128$ equivalent trees. This is because each transformation is a boolean choice. So, 127 trees are equivalent to this one tree.

4.7.18 Relational Algebra Trees

We can also construct a tree to represent a relational algebra expression.

Example What expression does the tree in Figure 49 represent?

We can create the following expression...

$$\pi_{title,year}(\sigma_{length \geq 100}(Movies) \cap \sigma_{studioName='Fox}(Movies))$$

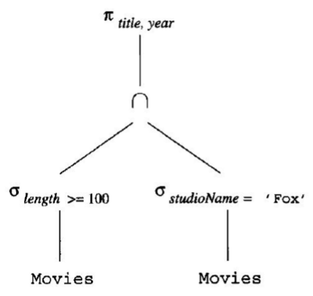


Figure 49: Expression Tree for A Relational Algebra Expression

4.7.19 Commutativity and Associativity in Relational Algebra

Observe that relational algebra also has a number of operators that are both commutative and associative.

Cross Product $R \times S = S \times R$ and $R \times (S \times T) = (R \times S) \times T$

Join $R \bowtie S = S \bowtie R$ and $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$

Union $R \cup S = S \cup R$ and $R \cup (S \cup T) = (R \cup S) \cup T$

Intersection $R \cap S = S \cup R$ and $R \cap (S \cap T) = (R \cap S) \cap T$

Therefore, we can... Swap left and right children of these operators in a relational algebra expression tree. Rotate any parents/children if the parent is one of these operators and the child is the same as the parent.

Note: For a theta join, the predicate must be well defined!

4.7.20 Laws Involving Selection

We have the following laws involving selection...

The Splitting Laws Conjunctions and disjunctions can be split into a composition of selections.

$$\sigma_{C_1\ AND\ C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R))$$
$$\sigma_{C_1\ OR\ C_2}(R) = (\sigma_{C_1}(R)) \cup_S (\sigma_{C_2}(R))$$

Pushing Selections Selection is distributive with respect to several other operators.

$$\begin{aligned}
\sigma_C(R \cup S) &= \sigma_C(R) \cup \sigma_C(S) \\
\sigma_C(R - S) &= \sigma_C(R) - S \\
\sigma_C(R - S) &= \sigma_C(R) - \sigma_C(S) \\
\sigma_C(R \times S) &= \sigma_C(R) \times S \\
\sigma_C(R \bowtie S) &= \sigma_C(R) \bowtie S \\
\sigma_C(R \bowtie_D S) &= \sigma_C(R) \bowtie_D S \\
\sigma_C(R \cap S) &= \sigma_C(R) \cap S
\end{aligned}$$

Unnamed Law Successive selections can be reordered in any permutation.

$$\sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_2}(\sigma_{C_1}(R))$$

4.7.21 Pushing Selections

Using the algebraic identities that permit pushing selections, how could we make the query plan in Figure 50 more efficient?

We could push the selection up the join and then back down both sides, since both have the attribute year. Then the join will have smaller operands. Figure 51 shows the alternate logical query plan.

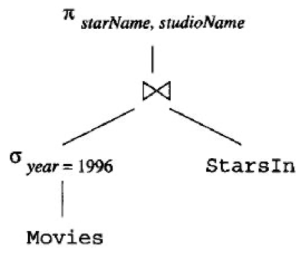


Figure 50: Logical Query Plan - A

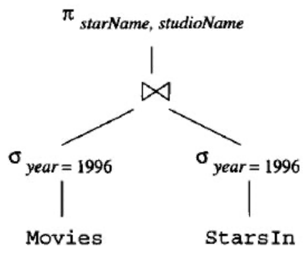


Figure 51: Logical Query Plan - B

4.7.22 Transforming SQL to a Logical Query Plan

We call a relational algebra expression tree a logical plan, since it indicates the order in which operators will be applied. Turning SQL into some plan is then as simple as converting it to relational algebra and then building an expression tree (in practice, this is more complex). Then we can apply transformations to produce a "better" query plan.

We can take any SQL query and convert it to relational algebra with the following naive approach:

- Put relations at the leaves
- Then join them
- The perform selections
- Then perform aggregation
- The apply HAVING predicates
- Then perform projection
- Then perform duplicate elimination
- Then perform sorting.

Note: Some children could be entire sub-queries!

4.7.23 Transforming SQL to a Logical Query Plan Examples

We can apply these laws and rules to now transform SQL to a logical query plan.

Example Convert the following SQL query into a logical query plan. Given R(a, b, c, d)...

```
SELECT a
FROM R
WHERE b < 100;
```

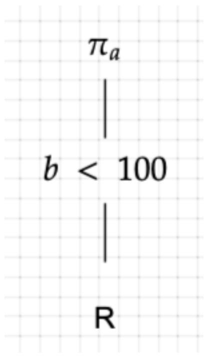


Figure 52: Logical Query Plan

Example Convert the following SQL query into a logical query plan. Given R(a, b, c, d)...

```
SELECT a
FROM R
WHERE a < 100;
```

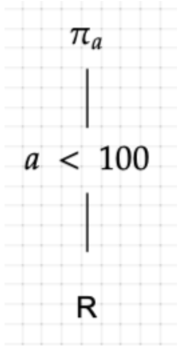


Figure 53: Logical Query Plan

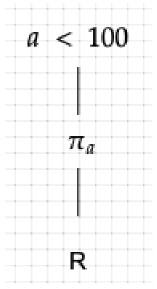


Figure 54: Logical Query Plan

Example Convert the following SQL query into a logical query plan. Given R(a, b, c, d) and S(d, e, f)...

```
SELECT *
FROM R, S
WHERE R.d = S.d
    AND R.d < 100;
```

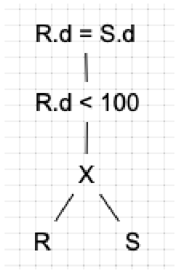


Figure 55: Logical Query Plan

Note: We have commutativity on X, 3 choices on selection, and a push selection on $R.d < 100$.

Example Convert the following SQL query into a logical query plan. Given R(a, b, c, d) and S(d, e, f)...

```
SELECT b, MAX(e)
FROM R
    NATURAL JOIN S
GROUP BY b
```

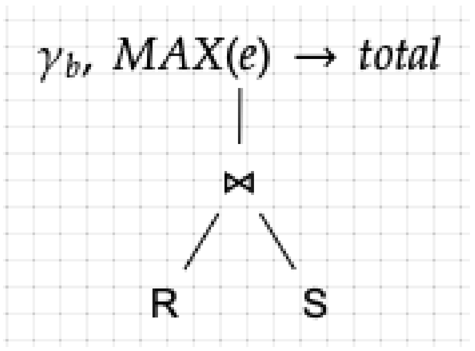


Figure 56: Logical Query Plan

Note: π is not necessary here as the rest of the attributes are no longer defined.

4.7.24 Summary

Algebraic expressions can be represented by (improper) binary trees. Numeric literals are leaf nodes and Operators are internal nodes.

Algebraic laws correspond to transforming on trees. Commutativity swaps the order of left and right children, Associativity rotates a tree, and Distributive laws push a node down below its children.

SQL can be converted into relational algebra and from relational algebra we can construct an expression tree, called a logical query plan. We can apply heuristics to make the logical query plan better, such as pushing selections as far down the tree as possible.

4.8 Cost Estimation and Physical Query Plans

Lesson Scope

In the previous lesson on Logical Query Plans, we learned how to transform a declarative SQL query into a partial order of operations defined by a parse tree. We also learned how to apply algebraic transformations to generate alternative logical query plans. However, we have not yet seen how to choose among those alternatives.

In this lesson, we learn how to ascribe an estimated cost to each logical query plan so that we can choose among alternatives quantitatively. We also learn how a physical database converts a logical query plan into a physical query plan that resolves remaining questions, such as which join algorithm should be used. The lesson covers §16.4–16.7 of Garcia-Molina et al. (2007).

Lesson Objectives

1. Given an operator and statistics about the input relation(s), you can estimate the size of the operator's output under standard statistical assumptions.
2. Given a logical query plan and statistics about the input relation(s) and memory, you can correctly estimate which operators require a materialisation of their results and which operators can pipe results to subsequent operators.
3. Given two equivalent logical query plans, you can use cost estimation to identify which is likely to incur fewer I/O's.
4. Given a physical query plan, you can provide plausible explanations for the design choices made in converting it from its corresponding logical query plan.

Delivery Method

We will follow the corresponding slide deck closely, which itself follows the textbook closely.

Preparation

This lesson assumes familiarity with relational algebra. You should be comfortable converting SQL queries into relational algebra and vice versa. It also assumes familiarity with disk-based join algorithms.

4.8.1 Selecting a Logical Query Plan

Consider the following SQL query...

```
SELECT R.x
FROM R JOIN S ON (R.x = S.x)
WHERE R.x = 87;
```

Now we will order these three equivalent logical query plans for the SQL below from fastest to slowest.

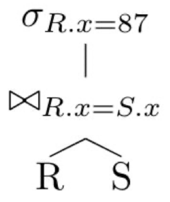


Figure 57: Plan A

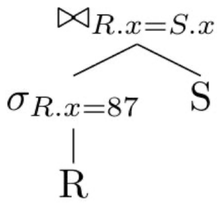


Figure 58: Plan B

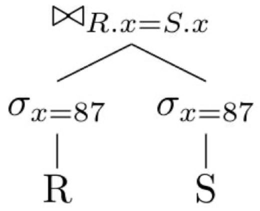


Figure 59: Plan C

We will assume **Plan C** $<$ **Plan B** $<$ **Plan A**, but how do we know that? And what if all R tuples have $R.x = 87$?

4.8.2 Probability of Events

Sample Space: The set of all possible outcomes of an experiment, denoted $\mathcal{S}$.

(Simple) Event: Any (size one) subset of outcomes contained in the sample space, $\mathcal{S}$.

Probability: An assignment of a number, $0 \leq P(A) \leq 1$ to an event A, which gives a precise measure of the chance that A will occur.

Example Say we toss a coin:

- One possible event is that it lands "heads up" which we could denote H.
- Another possible event is that it lands "tails up" which we could denote T.

Those are the only possible events, i.e., $\mathcal{S} = \{H, T\}$. If both options are equally likely, i.e., the coin is *fair*, then $P(H) = P(T) = 0.5$ (because the sum of probabilities of all possible events must sum to 1.0).

4.9 Independent Events

Independent Events: Events A and B are independent if $P(A|B) = P(A)$, i.e., knowing event B does not provide information about the probability of event A.

Multiplication Rule: A and B are independent if and only if $P(A \cap B) = P(A) * P(B)$.

Negation: The probability that event A did not occur is $1 - P(A)$.

Moreover, under independence $P(A \cup B) = 1 - (1 - P(A)) * (1 - P(B))$ (i.e., the probability that not neither both occurred).

Example What is the chance that if I toss a fair coin twice it will be H at least once?

We are asking, what is P(toss one is heads ∪ toss two is heads). We could more easily ask what is one minus the probability that it was never H?

Thus...

P(toss one is heads ∪ toss two is heads) = 1 - P(toss one is tails ∩ toss two is tails)
= 1 - 0.5(0.5) = 0.75.

4.9.1 Cost-Based Plan Selection

Consider again choosing between the three plans... We want to minimise I/O's, but all three plans are equivalent so they have the same output. However, the intermediate results may differ.

This, we can estimate the size of intermediate results and pick the plan for which the sum of intermediate results sizes is the smallest. BUT how do we estimate that?

4.9.2 Cost Estimation

We could consider the application of an operator to a tuple as a random event. If we can assign a probability to that event and assume that each tuple is an independent random variable, then the expected output would be the number of tuples multiplied by the probability. To estimate probabilities, the database (can) gather(s) some basic statistics.

- T(R): The number of tuples in relation R.
- B(R): The number of blocks occupied by relation R.
- V(R,x): The number of distinct values for attribute x in relation R.

Example What is T(R), V(R,name), and V(R,state) for R in Table 59?

We have T(R) = 5, since we have 5 tuples, V(R,name) = 5, since we have 5 unique names, and V(R,state) = 1, since we only have 1 value for state.

fips	name	state
1002	autauga	43
1004	balwin	43
1006	barbour	43
1008	bibb	43
1010	blount	43

Table 59: Relation R

4.9.3 Cost Estimation - Projections

How do we estimate the size of a projection? Bag semantics, we simply drop columns. The number of tuples is unchanged. The number of unique values is unchanged.

Example Consider the following ... $\pi_{(x,y)}(R)$.

$$\begin{tikzcd} \pi_{x,y} \arrow[d, dash] \\ \mathrm{R} \end{tikzcd}$$

```
T(R)   =  1000
V(R,x) =  10
V(R,y) =  100
```

Here, we have $\pi_{(x,y)}$ as $T(R) = 1000$ and R as $T(R) = 1000$.

4.9.4 Cost Estimation - Duplicate Elimination

How do we estimate the size of a duplicate elimination? Bag semantics, we simply drop columns. The number of tuples is unchanged. The number of unique values is unchanged.

Example Consider the following... $\delta(\pi_{(x)}(R))$.

$$\begin{tikzcd} \delta \arrow[d, dash] \\ \pi_x \arrow[d, dash] \\ \mathrm{R} \end{tikzcd}$$

```
T(R)   =  1000
V(R,x) =  10
```

Here, we have δ as $T(R) = 10$, $V(R,x) = 10$, π_x as $T(R) = 1000$, $V(R,x) = 10$, and R as $T(R) = 1000$, $V(R,x) = 10$.

4.9.5 Cost Estimation - Group By

How do we estimate the size of a grouping operator? It creates one group for every unique value. Therefore, it behaves like the duplicate elimination operator.

Example Consider the following... $\gamma_{(x)}(R)$.

$$\begin{tikzcd}\gamma_x \arrow[d, dash] \\ \text{R}\end{tikzcd}$$

```
T(R)   =  1000
V(R,x)  =  10
```

Here, we have $\gamma_{(x)}$ as $T(R) = 10$, $V(R,x) = 10$ and R as $T(R) = 1000$, $V(R,x) = 10$.

4.9.6 Cost Estimation - Selection

How do we estimate the size of a selection operator?

Make an assumption of uniformity for the distribution of values (i.e., tuple t in relation R has each possible value on attribute x with probability $\frac{1}{V(R,x)}$).

Assume each tuple is an independent random event sampled from the uniform distribution with replacement. Then, from *linearity of expectation*, we expect be $\frac{T(R)}{V(R,x)}$ tuples in the result for an exact equality predicate.

Example Consider the following... $\sigma_{(x=87)}(R)$.

$$\begin{tikzcd}\sigma_{x=87} \arrow[d, dash] \\ \text{R}\end{tikzcd}$$

```
T(R)   =  1000
V(R,x)  =  10
```

Here, we have $\sigma_{(x=87)}$ as $T(R) = 1000/10 = 100$, $V(R,x) = 1$ and R as $T(R) = 1000$, $V(R,x) = 10$.

Note: Multiple predicates can be treated as concurrent independent events.

Example Consider the following... $\sigma_{(x=87 \wedge y=42)}(R)$.

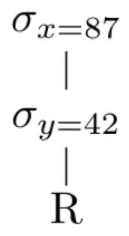
$$\begin{tikzcd}\sigma_{x=87 \wedge y=42} \arrow[d, dash] \\ \text{R}\end{tikzcd}$$

```
T(R)   =  1000
V(R,x)  =  10
V(R,y)  =  25
```

Here, we have $\sigma_{(x=87 \wedge y=42)}$ as $T(R) = 1000 * (1/10) * (1/25) = 4$, $V(R,x) = 1$, $V(R,y) = 1$ and R as $T(R) = 1000$, $V(R,x) = 10$, $V(R,y) = 25$.

Note: We can verify this with the splitting rule!

Example Consider the following... $\sigma_{(x=87 \wedge y=42)}(R)$.

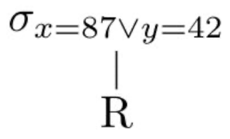


```
T(R)   = 1000
V(R,x) = 10
V(R,y) = 25
```

Here, we have $\sigma_{(x=87)}$ as $T(R) = 40/4 = 4$, $V(R,x) = 11$, $V(R,y) = 1$, $\sigma_{(y=42)}$ as $T(R) = 1000/25 = 40$, $V(R,x) = 10$, $V(R,y) = 1$, and R as $T(R) = 1000$, $V(R,x) = 10$, $V(R,y) = 25$.

Note: Disjunctions are treated like the union of random events.

Example Consider the following... $\sigma_{(x=87 \vee y=42)}(R)$.



```
T(R)   = 1000
V(R,x) = 10
V(R,y) = 25
```

Here, we have $\sigma_{(x=87 \vee y=42)}$ as $T(R) = 1000 * (1 - 1/25) * (1 - 1/10) = 864$, $V(R,x) = 10$, $V(R,y) = 25$ and R as $T(R) = 1000$, $V(R,x) = 10$, $V(R,y) = 25$.

4.9.7 Cost Estimation - Natural Join

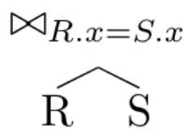
How do we estimate the size of a natural join operator?

Make two simplifying assumptions, since most joins are between a primary and a foreign key.

- *Containment of Value Sets*: Attribute values for one relation are a subset of those for the other; this implies the smallest set is trying to match values in the larger set.
- *Preservation of Value Sets*: An attribute not involved in the join will not lose any values.

Per containment, the probability that two tuples match on x is $\frac{1}{max(V(R,x),V(S,x))}$.

Example Consider the following... $R \bowtie_{(R.x=S.x)} S$.



```
T(R) = 1000, T(S) = 2000
V(R,x) = 10, V(S,x) = 2000
V(R,y) = 25, V(S,y) = 15
```

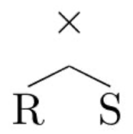
Here, we have $\bowtie_{(R.x=S.x)}$ as $T(R \bowtie S) = 1000 * 2000 * \frac{1}{max(10,2000)} = 1000$, $V(R \bowtie S, x) = 2000$, and $V(R \bowtie S, y) = 25$.

4.9.8 Cost Estimation - Cross Product

How do we estimate the size of a cross product?

Like a join, but with a match probability = 1.

Example Consider the following... $R \times S$.



```
T(R) = 1000, T(S) = 2000
V(R,x) = 10, V(S,x) = 2000
V(R,y) = 25, V(S,y) = 15
```

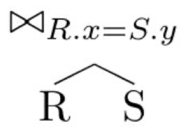
Here, we have $\times$ as $T(R \times S) = 1000 * 2000 * 1 = 2000000$, $V(R \times S, x) = 2000$, and $V(R \times S, y) = 25$.

4.9.9 Cost Estimation - Theta Join

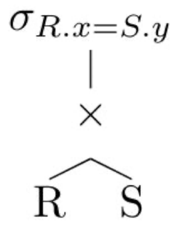
How do we estimate the size of a theta join?

We treat it like a cross product followed by a selection.

Example Consider the following... $R \bowtie_{(R.x=S.y)} S$.



```
T(R) = 1000, T(S) = 2000
V(R,x) = 10, V(S,x) = 2000
V(R,y) = 25, V(S,y) = 15
```



Here, we have $\sigma_{(R.x=S.y)}$ as $T(\sigma) = 2000000 * 1/2000 = 1000$, $V(\sigma, x) = 25$, $V(\sigma, y) = 25$ and $\times$ as $T(R \times S) = 1000 * 2000 * 1 = 2000000$, $V(R \times S, x) = 2000$, $V(R \times S, y) = 25$.

4.9.10 Calculate the Costs of Each Plan

How do we estimate the size of a theta join? We treat is like a cross product followed by a selection.

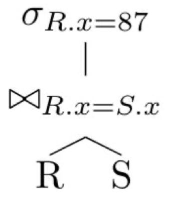


Figure 60: Plan A

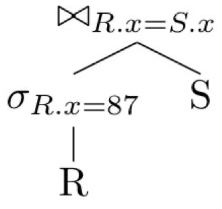


Figure 61: Plan B

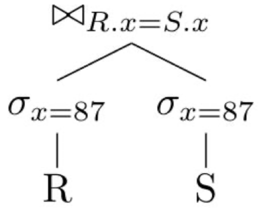


Figure 62: Plan C

Plan A: We have $T(R)T(S) * \frac{1}{2000} = 1000$. Thus, it will be 1000.

Plan B: We have $\sigma_{R.x=87} = 100$. Thus, it will be 100.

Plan C: We have $T(R)/V(R,x) = 100$ and $T(S)/V(S,x) = 1$. Thus, it will be 101.

4.9.11 Database Histograms

For very accurate estimates, we could also build histograms as an approximation to a probability density function (pdf). What is the size of the result, T(O), for

```
SELECT * FROM R JOIN S ON (R.x = S.x);
```

R.x	Count
$(-\infty, 25)$	99
$[25, 50)$	87
$[50, 75)$	7
$[75, \infty)$	0

S.x	Count
$(-\infty, 25)$	0
$[25, 50)$	0
$[50, 75)$	7
$[75, \infty)$	66

4.9.12 Selecting Physical Query Plans

After selecting a logical plan, with cost-based estimation, we still need to:

- Determine whether to materialise or pipe partial results.
- Select an algorithm for each operator.
- Decide how many buffers to allocate to each operator.

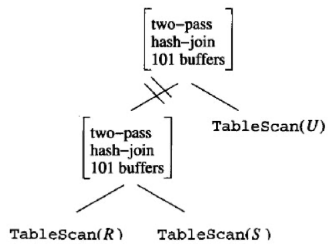


Figure 63: A Physical Plan

Observe: The selection of a query plan is part of the query time! We don't want to spend longer choosing a query plan than actually executing it.

4.9.13 Ordering Joins

Which is the best plan? Are these the only possibilities?

By convention: Left-hand argument is outer loop of join algorithm, called "build" relation and Right-hand argument is inner loop, called "probe" relation.

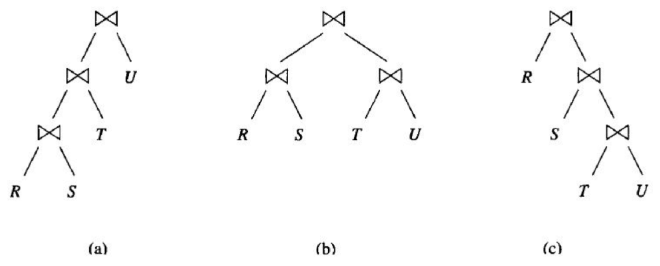


Figure 64: Ways to Join Four Relations

Heuristically, we want a left-deep tree; relations ordered left to right by increasing size.

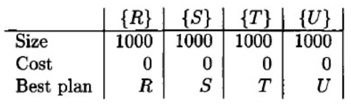
Example Calculate cost-based estimations; all relations have 1000 tuples.

$R(a,b)$	$S(b,c)$	$T(c,d)$	$U(d,a)$
$V(R,a) = 100$			$V(U,a) = 50$
$V(R,b) = 200$	$V(S,b) = 100$		
	$V(S,c) = 500$	$V(T,c) = 20$	
		$V(T,d) = 50$	$V(U,d) = 1000$

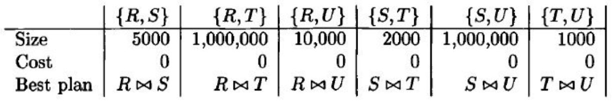
We see the difference between the best and the worst plan was $\frac{2000000}{3000} = 666\times$!

4.9.14 Ordering Joins - Greedy Algorithm

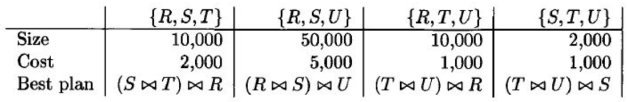
We will quickly build a good left-deep tree:



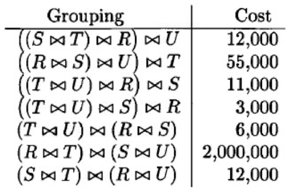
	$\{R\}$	$\{S\}$	$\{T\}$	$\{U\}$
Size	1000	1000	1000	1000
Cost	0	0	0	0
Best plan	R	S	T	U



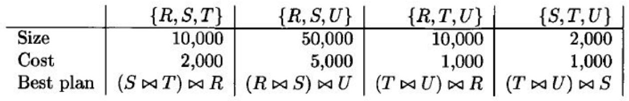
	$\{R,S\}$	$\{R,T\}$	$\{R,U\}$	$\{S,T\}$	$\{S,U\}$	$\{T,U\}$
Size	5000	1,000,000	10,000	2000	1,000,000	1000
Cost	0	0	0	0	0	0
Best plan	$R \bowtie S$	$R \bowtie T$	$R \bowtie U$	$S \bowtie T$	$S \bowtie U$	$T \bowtie U$



	$\{R,S,T\}$	$\{R,S,U\}$	$\{R,T,U\}$	$\{S,T,U\}$
Size	10,000	50,000	10,000	2,000
Cost	2,000	5,000	1,000	1,000
Best plan	$(S \bowtie T) \bowtie R$	$(R \bowtie S) \bowtie U$	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$



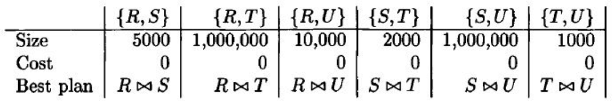
Grouping	Cost
$((S \bowtie T) \bowtie R) \bowtie U$	12,000
$((R \bowtie S) \bowtie U) \bowtie T$	55,000
$((T \bowtie U) \bowtie R) \bowtie S$	11,000
$((T \bowtie U) \bowtie S) \bowtie R$	3,000
$(T \bowtie U) \bowtie (R \bowtie S)$	6,000
$(R \bowtie T) \bowtie (S \bowtie U)$	2,000,000
$(S \bowtie T) \bowtie (R \bowtie U)$	12,000



	$\{R,S,T\}$	$\{R,S,U\}$	$\{R,T,U\}$	$\{S,T,U\}$
Size	10,000	50,000	10,000	2,000
Cost	2,000	5,000	1,000	1,000
Best plan	$(S \bowtie T) \bowtie R$	$(R \bowtie S) \bowtie U$	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$

1. Pick the pair of relations with the smallest estimated join size and join them.
2. While there are least two remaining relations:
   (a) Pick the relation whose estimated size with the join created so far is smallest and join it.

Example Using the example above...



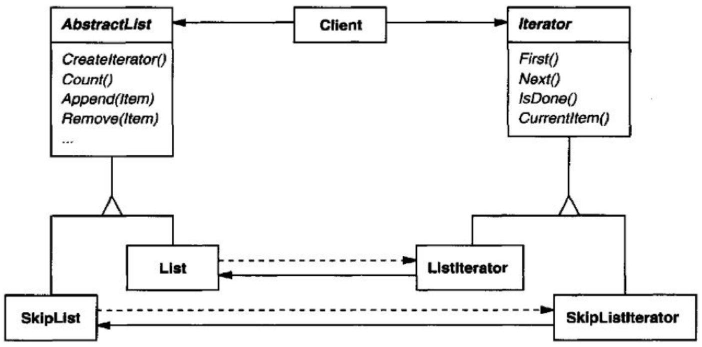
	$\{R,S\}$	$\{R,T\}$	$\{R,U\}$	$\{S,T\}$	$\{S,U\}$	$\{T,U\}$
Size	5000	1,000,000	10,000	2000	1,000,000	1000
Cost	0	0	0	0	0	0
Best plan	$R \bowtie S$	$R \bowtie T$	$R \bowtie U$	$S \bowtie T$	$S \bowtie U$	$T \bowtie U$

We will start with $T \bowtie U$, next join $(T \bowtie U) \bowtie S$, and we end up with the min cost join order $((T \bowtie U) \bowtie S) \bowtie R$.

4.9.15 Background: Iterator Pattern

Intent: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Takes the responsibility for access and traversal out of the list object and put it into an iterator object. Each container has a factory method to create an iterator. Possible to create different iterators with different iteration logic (e.g., ForwardIterator and ReverseIterator). It is then possible to *iterate* the contents of the container with the Iterator interface alone.



4.9.16 Pipelining vs Materialisation

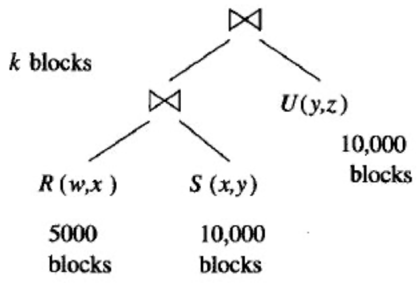
To compose operators, we have two options...

- Use the iterator pattern on the output of the first operator.
- Materialise the result of the first operator as a temporary table.
  - This uses more memory, which may reduce concurrency.
  - If it doesn't fit in memory, it costs I/O's to write the temporary result to disk and read it back again.

When we don't need to write the temporary result to disk, we call it pipelining. Otherwise, the strategy is called materialisation and is denoted by a double-line on the query plan.

Example Should we "materialise" this result?

There are 101 blocks of memory available.



4.9.17 Scan Algorithms

We have seen several choices of different join algorithms. Let us also consider reading tables and performing selections:

- **Operators for Leaves**:

  - *TableScan(R)*: We simply read in R from disk in arbitrary order.
  - *SortScan(R,L)*: Read tuples of R from disk in order L (e.g., if there is an index).
  - *IndexScan(R,C)*: Tuples matching condition C are retrieved using an index on R.
  - *IndexScan(R,A)*: All tuples of R are retrieved using an index on attributes R.A.

- **Operators for Selection**:
  - *Filter(C)*: Retain only those tuples from the sub-tree in which C holds.

We will generally need it take into account whether indexes exist on the attributes that we want and what type of indexes those are (e.g., must have B+-tree not a hash index for an IndexScan(R,C) where C is an inequality.

4.9.18 Final Query Plan Notation

Note: You can write EXPLAIN in front of any SQL query and it will tell you the query plans it is considering.

4.9.19 Summary

Producing logical query plans is just the first step.

We can estimate the size of intermediate results by tracking statistics. We can then select a logical plan with minimal expected size.

To convert the logical plan into a physical plan, we consider...

- Which algorithms are available to us for each operator.
- Make concrete choices about commutativity (which is the left-hand operand).
- Make choices about how many buffers we need and whether intermediate results can be piped or must be materialised - and, if materialised, whether that can be done in memory.

4.10 Midterm 04

4.10.1 Scope

This exam covers the Storage and Query Processing unit of the course, including §13–17 of the textbook. The exam assesses your knowledge of database storage and query processing, particularly that:

- given the content of a set of disks, some of which have failed, you can apply a specified RAID scheme to completely restore data from the failed disks.
- given a database log and the current contents on disk of the database, you can restore the database to the most up-to-date consistent state possible in which all transactions are atomically committed or aborted.
- given a B+-tree or R-tree and a query, you can indicate the exact cost in the I/O model of executing that query.
- given a (possibly empty) B+-tree or R-tree and a list of keys to insert and/or delete, you can draw the tree that results from that sequence of modifications.
- given a join algorithm, the contents of two disk-resident tables, the system block size and the size of main memory, you can calculate the total number of I/O's required by the join algorithm to write the join result back to disk.
- given a SQL query, you can produce several equivalent logical query plans by means of naive translation and by pushing operators correctly up or down the query plan.
- given two equivalent logical query plans and relevant histograms, you can determine which is likely to be more efficient by means of heuristic cost estimation.

4.10.2 Format

This *closed-book* exam is written in-person. If you do not hand in this exam, the weight will be shifted to Assignment 4. The practice exam below is highly predictive of the style of questions that you can expect. You have up to fifty-five minutes to write the exam, after which we will take a ten minute break and then have a short lecture to wrap up the course.

5 Wrap-Up

Overview

At this point, you are proficient in relational databases. Thanks for making it to this point!

In this short course outro, you will:

- recall the units of the course and how they fit together.
- consider important aspects of modern-day data management that were not covered in this course.
- reflect on how this course connects to other courses in the undergraduate curriculum.

Module-Level Intended Learning Outcomes

Upon completion of this short introductory module, you can:

- when the desire strikes, plan a concrete path to further your learning on subject matter related to this course.

References

Harrison (2015) [4]. Next Generation Databases: NoSQL, NewSQL and Big Data. Apress. Selections.

5.1 Course Outro

Lesson Scope

In this lesson, we will outline the importance of ACID-compliance and the importance of abstraction.

We will also go over questions that we did not have time to answer.

Lesson Objectives

1. Having taking this course, you can: Reflect on what you have learned and how you can further grow that knowledge.

Delivery Method

We will follow the corresponding slide deck closely.

5.1.1 Role of Databases in Applications: ACID

We learned that, compared to a file system, databases provide ACID properties.

If the data used by your application needs any of these properties (or especially all of them), then an ACID-compliant database like MySQL is probably a good choice.

Atomicity With transactions and logging, we can ensure that either same things happen altogether or none of them do, even in the presence of unexpected external events like a power outage.

An example would be Transactions in SQL. For example...

```
BEGIN TRANSACTION;

UPDATE BankAccount
SET value = value + 100
WHERE account_id = 42;

UPDATE BankAccount
SET value = value − 100
WHERE account_id = 87;

COMMIT;
```

Consistency With constraint checking and logging, we can maintain "consistency", i.e., that the data will never enter a state that violates the rules specified in the schema.

An example would be altering a table in SQL to have a unique constraint. For example...

```
ALTER TABLE R MODIFY x INT UNIQUE;

INSERT INTO R(x) VALUES(42);

INSERT INTO R(x) VALUES(42);

ERROR 1062 (23000): Duplicate entry '42' for key 'R.x'.
```

Isolation With isolation levels, we can control the notion of isolation, i.e., what things are allowed to happen at the same time, which is foundational to enabling multiple users to concurrently access the database.

Durability With non-volatile storage, a switch to an I/O model of computation, and RAID redundancy, we can guarantee that once committed, data will not be lost.

An example would be RAID SCHEME 4. For example... Consider the following layout.

Disk 1: 0111 0101
Disk 2: 1010 0111
Disk 3: 1101 0010

We want to restore Disk 3, so we use Disk 1 and Disk 2. We will calculate the following $01110101 \oplus 10100111 = 11010010$ and we have now restored Disk 3.

5.1.2 Role of Databases in Applications: Abstraction

We learned that databases isolate layers of abstraction:

Generally, we build an application layer that interfaces with the logical layer of the database. Thus, our C# or Java application only needs to worry about converting objects to tables and vice versa, not about the mechanics of data storage or thread safety on data writes or any physical implementation matters. This can make developers much more productive and make code much more safe.

You don't need to understand how a database works in order to be a SQL programmer.

You don't need to understand how to normalise relations to improve data consistency in order to create precise, formal, conceptual descriptions of what a database needs to contain and the relationship between those data.

Examples: Requirements, Conceptual Design, Logical Design, Physical Design.

5.1.3 Unanswered Questions

Question A DBMS is comprised of a series of independent modules that together operate as a complete system with direct access to the underlying file system.

We did not discuss much the **modular design of complex systems and how they interface**.

Question A DBMS supports concurrent access.

We did not discuss the **physical implementaton of isolation guarantees**, only how to specify them for a transaction.

Future Topics: heavy-weight solutions for critical sections like locks, reasoning about correctness for concurrent and asynchronous computing, non-blocking techniques for concurrency, design patterns for parallel algorithms, models of computation for parallel algorithms, vectorisation and data-oriented programming.

Question A DBMS typically uses RAID schemes for durability. We prefer not to have "data disks" and "back-up disks" because the write traffic to back-up disks becomes a bottleneck. Thus, production databases are, pretty much by default, distributed systems.

We did not discuss properties of **availability and partition tolerance** which are central in modern, globally distributed database systems.

Future Topics: how to manage the connection between nodes over a network, how to manage system that is distributed across multiple nodes, how to use distributed frameworks like Apache Spark and TensorFlow to answer queries over distributed data.

Note: If you are interested in becoming a database architect, I believe these topics are required knowledge today.

Question A DBMS maintains data consistency automatically through normalisation; queries join tables to construct a complete view of the data, but we also learned that joins are one of the most expensive operators that we can execute.

We did not discuss when we might be interested in the trade-off between query performance and data consistency guarantees that we can chose with **deliberate denormalisation** of the data.

Future Topics: NoSQL models are based on recoupling the physical and logic layers of the database and denormalising data to improve query response time, write speed, congruence with object-orientation, and availability. Examples model data as graphs, XML document trees, and as dictionaries.

Note: You are very likely to be exposed to these alternative NoSQL models in industry in Victoria.

5.1.4 Engaging State-of-the-Art

The field of databases evolves as often as data evolves...

You now have a solid enough foundation that you can understand some research in the field.

- The flagship ACM conference: SIGMOD (special interest group on the management of data). More information here: https://dblp.org/db/conf/sigmod/index.html.
- An independent and equally reputable alternative: VLDB (vary large data bases). More information here: https://dblp.org/db/conf/vldb/index.html.
- The top IEEE conference on the topic is ICDE (International Conference on Data Engineering). More information here: https://dblp.org/db/conf/icde/index.html.

Facebook, Google, Microsoft, LinkedIn, etc., consistently publish their latest techniques in these venues.

6 Resources

Setting up an RDMBS

Resources to help you set up a local instance of a database management system like MySQL.

MySQL On Non-Linux Systems

This course will officially support MySQL on Linux-based systems, particularly Ubuntu. However, below you will find some student-supplied tips for supporting Mac OS and Windows, particularly to get around the secure-file-priv error that you are likely to encounter if you try to import data files.

Mac OS (e.g., Catalina)

Because I have just tested this approach myself, I recommend installing MySQL through home brew. If you don't yet have home brew, you will need to install that first. As per the official site, https://brew.sh/:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Once you have installed homebrew, you can install mysql:

```
brew install mysql
```

I found it easier to set up the root password by next using the command:

```
mysql_secure_installation
```

You can then connect to the local MySQL server in the same way as in my Linux demos:

```
mysql -u root -p
```

At this point, you can use MySQL but you will still have challenges with the LOAD DATA INFILE command because of the secure-file-priv default setting. You can disable this, as per https://stackoverflow.com/a/43286928/2769271, by creating a new file in your home directory called .my.cf with the following contents:

```
[mysqld_safe]
[mysqld]
secure_file_priv=""
```

I then restarted the MySQL server for good measure:

```
brew services restart mysql
```

Windows

Here is what I did:

1. Located the my.ini file in the $C : \backslash ProgramData \backslash MySQL \backslash MySQL\ Server$ 8.0 directory.
2. Edit that file so that it looks like the image below with the second line commented out and the third line added:
    - # Secure File Prive.
      # secure-file-priv="C:/ ProgramData / MySQL / MySQL Server 8.0 / Uploads"
      secure-file-priv=""
3. Move the imdb-snapshot folder [my annotation: this is where the student had the data that he wanted to import] into the following directory: C: / ProgramData / MySQL / MySQL Server 8.0 / Data.
4. Create the table in MySQL.
5. Load the data into the table in MySQL using the following type of query:
    - LOAD DATA INFLINE imdb−snapshot/ratings.tsv
      INTO TABLE ratings
      FIELDS TERMINATED **BY** /t
      IGNORE 1 LINES;

After I did those steps, I managed to properly load the data into MySQL and query the tables accordingly. I was using the MySQL Command Line Client the whole time, as I did not find a need to open MySQL in a standard terminal window.

MySQL and DBeaver Setup

In the lectures, we will use the Terminal exclusively to practise writing SQL, but you could also choose to use DBeaver to connect and modify your db server.

MySQL Setup and Terminal Usage

- Follow MySQL Installation Excerpt to install and configure MySQL on your machine. You might need to do additional configuration on your environment; See Installing MySQL - Windows Linux Mac for more details. (If you are using a Mac, Homebrew is a great option).
- After successful installation, open the Terminal and establish the connection to the db server. The command is `services start mysql` but depending on your platform, you might need to add keywords such as `sudo services start mysql` or `brew services start mysql`.
- Type `mysql_secure_installation` to setup the server. The following is a suggested setup example:
  - **VALIDATE PASSWORD PLUGIN**: NO
  - **Remove Anonymous User**: YES
  - **Disallow Root Login from Remote**: NO
  - **Remove the Test Database**: YES
  - **Reload Privilege Tables Now**: YES
- Type `mysql -uroot -proot` to login into the SQL server.
- Now you can input SQL queries into the db directly. Some common commands include:
  - SHOW DATABASES;
  - **CREATE** DATABASE testdb;
  - exit

DBeaver Setup

To setup, download and install DBeaver first.

- Click on the plugin icon in the top left corner of DBeaver. Then, specify the details of your db server.
- You could then click on the SQL button to write SQL queries.
- Alternative, the GUI also provides wizards to help execute certain actions. For example, you could visually see how the table looks like, and you could right click on your db in the Navigator to create a table directly.

Remove MySQL - Mac OSX

To remove MySQL completely on Mac OSX you might need to use the following: GitHub Gist.

References

[1] C. Batini, S. B. Navathe, and S. Ceri, *Conceptual Database Design: An Entity-Relationship Approach*, English, 1st Edition. Addison-Wesley, Aug. 1991, ISBN: 978-0-8053-0244-8.

[2] H. Garcia-Molina, J. Ullman, and J. Widom, *Database Systems: The Complete Book*, English, 2nd Edition. Upper Saddle River, N.J: Pearson, Jun. 2008, ISBN: 978-0-13-187325-4.

[3] E. Chance, *SQL Transaction Isolation Levels Explained*, English, Jun. 2019. [Online]. Available: https://elliotchance.medium.com/sql-transaction-isolation-levels-explained-50d1a2f90d8f (visited on 11/30/2022).

[4] G. Harrison, *Next Generation Databases: No SQL, NewSQL and Big Data*. Apress, 2015.

[5] M. T. Goodrich and R. Tamassia, *Algorithm Design and Applications*, English. Oct. 2014, ISBN: 978-1-118-33591-8.

[6] Oracle, *MySQL :: MySQL 8.0 Reference Manual :: 2 Installing and Upgrading MySQL*, English, 2022. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/installing.html (visited on 11/30/2022).

[7] S. Chester and Y. Zhao, *UVic CSC 370 Course Outline*, English, Sep. 2022. [Online]. Available: https://heat.csc.uvic.ca/coview/course/2022091/CSC370 (visited on 11/30/2022).