The I/O Model of Computation (13.3.1)

- 1 processor, 1 disk controller, 1 disk.

Dominance of I/O Cost: The time taken to perform a disk access is much larger than the time likely to be used manipulating that data in main memory. Thus, the number of block accesses (Disk I/O's) is a good approximation to the time needed by the algorithm and should be minimized.

Disk Failures (13.4)

- Intermittent Failure: read / write is unsuccessful, but repeated tries are successful.
- Media Decay: bit(s) are permanently corrupted (read).
- Write Failure: Cannot write, power outage.
- Disk Crash: Entire disk unreadable permanently.

Checksums: additional bits for each sector.
- Reduces the probability of missing a bad read.

Parity: Odd (1) / Even (0). The number of 1's among a collection of bits and their parity bit is always even. An odd parity indicates the presence of an error. Chance of missing an error $1/2^n$; n is parity bit.

Stable Storage: (1) Write the value of X into X_L. Check that the value has status "good". If not, repeat the write. If after a set number of write attempts, we have not successfully written X onto X_L, assume that there is a media failure in this sector. A fix-up such as substituting a spare sector for X_L must be adopted. Repeat (1) for X_R. (Alternate X_L and X_R)

Media Failures: If X in X_L and X_R, if X_L or X_R is permanently unreadable we can read from the other.

Write Failure: failure when writing X_L. X_L is bad. X_R is good. failure after writing X_L. X_L is good. Copy X_L to X_R.

RAID: Redundant Arrays of Independent Disks

Level 1 – Mirror: data disk and redundant disk

Level 4 – Parity Blocks: Modulo-2 Sum

D1	11110000
D2	10101010
D3	00111000
D4	01100010

Update D2: 11001100 ⊕
Old D2: 10101010
Parity: 01100110
Old D4: 01100010 ⊕
New D4: 00000100

- The bit in any position is the modulo-2 sum of all the bits in the corresponding positions of all the other disks.

Level 5: 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.

Level 6 – Hamming Code:

Disk #	1	2	3	4	5	6	7
	Data				Redundant		
	1	1	1	0	1	0	0
	1	1	0	1	0	1	0
	1	0	1	1	0	0	1

Write: Update D2, we update D5 and D6.

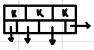
OD2 ⊕ ND2 = PD2, PD2 ⊕ D5 and PD2 ⊕ D6.

Recover (up to 2 disks): D2 and D5 are lost, row 2 has D2 as 1 and D5 as 0, so D1, D4, and D6 to recover D2.

D2 = D1 ⊕ D4 ⊕ D6 and D5 = D1 ⊕ D2 ⊕ D3. $2^K - 1$, K are redundant, $2^K - K - 1$ are data disks. Construct matrix with all possible columns of K 0's and 1's.

B-Trees (14.2) (B+ – Tree)

All paths from the root to a leaf have the same length. Root, Intermediate, Leaves

n search-key values and n+1 pointers. 

B = 4096 bytes, Keys 4 bytes, pointers 8 bytes: $4n + (8n+1) \leq 4096$, n = 340.

Lookup: Search for K recursively starting at root and ending at a leaf. (Height)

Range Queries: [a, b]. Lookup Key a (a or greater and not > b) (Height + output)

Insertion: [1|2|3] → [1|2|] [3|4|], [3| |] K into the parent; $\lceil (n+2)/2 \rceil$ $\lfloor (n+2)/2 \rfloor$

Deletion: Lookup and delete. Rebalance the B+-Tree.

Efficiency: Neglect I/O cost of re-organization. 3 disk I/O's.

Layout: R has T = 2048 with 320 bytes and B = 4096 bytes.

Block contains $\lfloor 4096/320 \rfloor$ = 12 records. R spans $\lceil 2048/171 \rceil$ = 171 blocks.

The Computation Model for Physical Operators (15.1.3)

Arguments of any operator are found on disk, result is left in main memory.

Parameters for Measuring Costs (15.1.4)

- M denotes the estimate number of main-memory buffers available.

Three parameter families: B, T, and V.
- B(R), Blocks, R is clustured.
- T(R), tuples in R: ratio T/B. fit in a block.
- V(R, a), number of distinct values of the column for a in R. $\delta(\pi_a(R))$

Tuple-Based Nested-Loop Join (15.3.1)

$R(x,y) \bowtie S(y,z)$: T(R)T(S) disk I/O's.

```
FOR each tuple s in S DO
  FOR each tuple r in R DO
    If r and s join to make a tuple t THEN
      Output t;
```

(Index R to lower cost).

An Iterator for Tuple-Based Nested-Loop Join (15.3.2)

Allows us to avoid storing intermediate relations on disk. R ⋈ S.

Block-Based Nested-Loop Join Algorithm (15.3.3)

B(R)(B(S)/2) + B(S) disk I/O's.

```
FOR each block br in R DO
  FOR each block bs in S DO
    FOR each tuple r in br DO
      FOR each tuple s in bs DO
        If r.join_Keys == s.join_Keys THEN
          EMIT r JOIN s;
```

Analysis of Nested-Loop Join (15.3.4)

Disk I/O's B(S)(M−1+B(R))/(M−1) or B(S)+(B(S)B(R))/(M−1). Any B(R), B(S) and M. (S is the smaller relation)

Approximation: B(S)B(R)/M. Assuming B(R), B(S), and M are large. (but M is smallest)

If B(S) ≤ M−1, the nested-loop join becomes identical to the one pass join algorithm.

Summary of Algorithms (15.3.5)

Operators	Approximate M required	Disk I/O
σ, π	1	B
γ, δ	B	B
∪, ∩, −, ×, ⋈	min (B(R), B(S))	B(R) + B(S)
⋈	any M ≥ 2	B(R)B(S) / M

The Hash-Join Algorithm (15.5.5)

Compute $R(x,y) \bowtie S(y,z)$ using a two-pass, hash-based algorithm. Use the join attributes y as the hash Key. Buckets R_i and S_i; i. A one-pass join of all pairs of buckets to complete the algorithm.

Disk I/O's 3(B(R) + B(S)), Approx: min(B(R), B(S)) ≤ M^2.

Joining by Using an Index (15.6.3)

R is clustured: Read B(R) blocks to get all tuples of R.

R not clustured: Up to T(R) disk I/O may be required.

Each tuple t of R: T(S)/V(S,y) tuples of S. If S has a nonclustured index on y, to read S: T(R)T(S)/V(S,y), index clustured: T(R)B(S)/V(S,y) T(R)(max(1, B(S)/V(S,y)))

The Query Compiler (16)

SQL parsed into structured tree, parse tree into an expression tree of relational algebra (logical query plan), turned into physical query plan.

Query → Parser → Preprocessor → Logical query plan generator → Query rewriter → Preferred logical query plan.

The preprocessor is responsible for semantic checking.
- Check Relation Uses: Every relation mentioned in a FROM-Clause must be a relation or view in the current schema.
- Check and Resolve Attribute Uses: Every attribute must come from some relation in the current scope. (check Ambiguity).
- Check Types: All attributes must be of a type appropriate to their use.

SELECT A FROM R WHERE C;

π_A —— σ_C —— R (Left to Right)

Algebraic Laws for Improving Query Plans (16.2)

Commutative Law: Order does not matter (addition, multiplication).

Associative Law: Group operators by two left or right ((x+y)+z = x+(y+z)).

Operators that are both associative and commutative: (Laws hold for bags and sets)
- R × S = S × R; (R × S) × T = R × (S × T).
- R ⋈ S = S ⋈ R; (R ⋈ S) ⋈ T = R ⋈ (S ⋈ T).
- R ∪ S = S ∪ R; (R ∪ S) ∪ T = R ∪ (S ∪ T).
- R ∩ S = S ∩ R; (R ∩ S) ∩ T = R ∩ (S ∩ T).

Cummutative Operator: $R \bowtie_C S = S \bowtie_C R$.

Laws Involving Selection: "push selections down the tree".

Splitting Laws $\sigma_{C_1 \text{ AND } C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R))$
- $\sigma_{C_1 \text{ OR } C_2}(R) = (\sigma_{C_1}(R)) \cup_S (\sigma_{C_2}(R))$; R is a set.

Union: $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$.

Difference: $\sigma_C(R-S) = \sigma_C(R) - \sigma_C(S)$, $\sigma_C(R-S) = \sigma_C(R) - S$.

Selection can be pushed down, but only if R has all attr. of σ_C.
- $\sigma_C(R \times S) = \sigma_C(R) \times S$.
- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$.
- $\sigma_C(R \bowtie_D S) = \sigma_C(R) \bowtie_D S$.
- $\sigma_C(R \cap S) = \sigma_C(R) \cap S$.

Pushing Selections: move selection as far up and then pushed down all possible branches.

Laws Involving Projection: "pushed down".

Consider E → x, then E is input and x is output.
- $\pi_L(R \bowtie S) = \pi_L(\pi_M(R) \bowtie \pi_N(S))$; L is input and M, N are join.
- $\pi_L(R \bowtie_C S) = \pi_L(\pi_M(R) \bowtie_C \pi_N(S))$
- $\pi_L(R \times S) = \pi_L(\pi_M(R) \times \pi_N(S))$
- $\pi_L(R \cup_B S) = \pi_L(R) \cup_B \pi_L(S)$
- $\pi_L(\sigma_C(R)) = \pi_L(\sigma_C(\pi_M(R)))$.

Laws About Joins and Products: Apply right to left.
- $R \bowtie_C S = \sigma_C(R \times S)$.
- $R \bowtie S = \pi_L(\sigma_C(R \times S))$.

Laws Involving Duplicate Elimination: Moving down.
- $\delta(R) = R$ if R has no duplicates. (i.e., primary key, etc.)
- $\delta(R \times S) = \delta(R) \times \delta(S)$.
- $\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$.
- $\delta(R \bowtie_C S) = \delta(R) \bowtie_C \delta(S)$.
- $\delta(\sigma_C(R)) = \sigma_C(\delta(R))$.
- $\delta(R \cap_B S) = \delta(R) \cap_B S = R \cap_B \delta(S) = \delta(R) \cap_B \delta(S)$.

Laws Involving Grouping and Aggregation: No stated laws, but γ absorbs a δ. $\delta(\gamma_L(R)) = \gamma_L(R)$.
- $\gamma_L(R) = \gamma_L(\pi_M(R))$
- $\gamma_L(R) = \gamma_L(\delta(R))$; γ_L is dup.-impervious.

Parse Trees to Logical Query Plans

1. Replace nodes by operators of relational algebra.
2. Relational algebra expressions into expressions most efficient for physical query plan.

Estimating the Cost of Operations.

Cost-based enumeration: least estimated cost.

Estimate the size of a projection: grow or shrink.

Estimate the size of a selection: Let $S = \sigma_{A=c}(R)$; A is an attr. of R and c is a constant. $T(S) = \frac{T(R)}{V(R,A)}$

If $S = \sigma_{a<c}(R)$. $T(S) = T(R)/3$.

If $S = \sigma_{a \neq 10}(R)$. $T(S) = T(R)$. (or. $T(S) = T(R) \cdot \frac{(V(R,a)-1)}{V(R,a)}$)

If $S = \sigma_{C_1 \text{ OR } C_2}(R)$, then R has n tuples, m of which satisfy C..

$T(S) = n(1-(m_1/n)(1-m_2/n))$.

Estimate the size of a Join: (Natural Join)
- Containment of Value Sets: If R and S are two relations with an attribute y, and V(R,y) ≤ V(S,y), then every y-value of R will be a y-value of S.
- Preservation of Value Sets: If A is an attr. of R but not of S, then V(R ⋈ S, A) = V(R, A).
- T(R ⋈ S) = T(R)T(S)/max(V(R,y), V(S,y)).

Estimate Difference (T(R) − T(S)/2).

Cost-Based Plan Selection: Number of Disk I/O's.

Histogram: of the values for a given attribute.

Equal-width: A width w is chosen, along with a constant v_0.

Equal-height: Common "percentiles".

Most-frequent-values: list of common values.

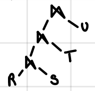
- To compare two plans, we add the estimated sizes of all the nodes except the root and the leaves.

Choosing an Order for Joins: Left argument of the join is the smaller relation and store it in main memory (build) Right argument (probe), is read a block at a time and its tuples are matched in in main memory with those of the build relation.

□ Nested-Loop Join: left argument is outer.

□ Index-Join: right argument has th index.

Left-Deep Join Trees:



$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$.

Greedy Algorithm for Selecting a Join Order

Make one decision at a time about th order of Joins and never backtrack or reconsider descisions.

Pipelining Versus Materialization

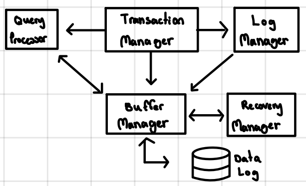
Materialization: Each intermediate relation is materialized on disk. (store th result of each operation on disk until it is needed by another operation)

Pipelining: Tuples produced by one operation are passed directly to th operation that uses it, without storing th intermediate tuples on disk. Save disk I/O's. (unary operations, selection, projection).

Physical Query Plans: TableScan(R), SortScan(R,L), IndexScan(R,C), IndexScan(R,A).

More About Transactions (17.1.2)

Transaction Manager: log records, recovery.



The Correctness Principle: If a transaction executes in th absence of any other transactions or system errors, and it starts with th database in a consistent state, th th database is also in a consistent state when th transaction ends.

Primitive Operations of Transactions

INPUT(X), READ(X,t), WRITE(X,t), OUTPUT(X).

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t := t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t := t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Undo Logging: Makes repairs to th database state by undoing th effects of transactions that may not have completed before th crash.

Flush-Log: log itself from main memory and copied to disk by an operation.

<START T>: Transaction T has begun.

<COMMIT T>: Transaction T was "successful."

<ABORT T>. Transaction T wasn't "successful."

<T, X, v>: Transaction T has changed database element X, and its former value was v. (Occurs in memory, not disk (i.e., after WRITE)).

Rules: The log records indicating changed db elements. The changed db elements themselves. The COMMIT log record.

(i.e. FLUSH → OUTPUT → COMMIT → FLUSH)

Recover: Divide th transactions into committed and uncommitted. A log record of <COMMIT T> means T is on disk. A <START T> without a commit needs to be fix by undoing. All changes must be reset to old v. After restoring backwards, log <ABORT T> and flush log.

Checkpointing: 1. Stop accepting new transactions. 2. Wait until all currently active transactions commit/abort and have log. 3 Flush th log to disk. 4. Write a log <CKPT>, and flush again. 5. Resume accepting transactions.

Nonquiescent Checkpointing: 1. Write a log record <START CKPT(T₁,...,Tk)> and flush th log. Here, $T_1, \ldots, T_k$ are th names or identifiers for all th active transactions. 2. Wait untill all of $T_1, \ldots, T_k$ commit/abort, but do not prohibit other transactions from starting. 3. When all of $T_1, \ldots T_k$ have completed, write a log record <END CKPT> and flush log.

Redo Logging: ignores incomplete transactions and repeats the changes made by commited transactions. It requires that th commit record appear on disk before any changed values reach disk.

Rules: <T, X, v>: Transaction T wrote new value v for database element X. (redo rule: Write-ahead logging rule). The log records indicating changed database elements. The COMMIT log record. Th changed database elements.

<T, A, 16>, <COMMIT T>, FLUSH, OUTPUT(A).

Unless th log has a <COMMIT T> record, we know that no changes to th database made by Transaction T have been written to disk. Thus, incomplete transactions may be treated during recovery as if they had never occured.

Recover: 1. Identify the committed transactions. 2. Scan th log forward from th beginning for each log record <T, X, v> a) If T is not a committed transaction, do nothing. b) If T is committed, write value v for database element X. 3. For each incomplete transaction T, write an <ABORT T> record to the log and flush th log.

Checkpointing: 1 Write a log record <STARTT CKPT(T₁,...Tk)> where $T_1, \ldots T_k$ are all th active (uncommitted) transactions, and flush log. 2. Write to disk all db elements that were written to buffers but not yet to disk by transactions that had already committed when th START CKPT record was written to th log. 3. Write an <END CKPT> record to th log and flush th log.

Recovery Redo: If <END CKPT>, we know that every value written by a transaction that committed before th corresponding <START CKPT(T₁,...,Tk)> has had its changes written to disk, so we can ignore. All T_i's and transactions after th beginning of th checkpoint need to be redone. In searching the log, we do not have to look further back than the earliest of of th <START T_i> records. If <START CKPT(T₁,...,Tk)> then we search back to the previous <END CKPT> record, find its matching <START CKPT(S₁,...Sm)> record, and redo all those committed transactions that either started after that START CKPT or among th S_i's. (There could be a START CKPT record that has no matching <End CKPT> record. Therefore, need not just for th previous START CKPT, but first for an <END CKPT> and the prev. START).

Undo/Redo Logging: The update log record that we write when a database element changes values has four components. <T, X, v, w>; former value was v, and its new value is w.

Rule: Before modifying X on disk, record <T, X, v, w> appear on disk. <T, A, 8, 16>, <T, B, 8, 16>, FLUSH, OUTPUT(A), <COMMIT T>, OUTPUT(B). A <COMMIT> record flushed ASAP.

Recovery: 1. Redo all th committed transactions in th order earliest-first, and 2. Undo all th incomplete transactions in th order latest-first

Checkpointing: 1. Write a <START CKPT(T₁,...,Tk)> record to th log, where $T_1, \ldots, T_k$ are all th active transactions, and flush th log. 2. Write to disk all th buffers that are dirty; i.e., they contain one or more changed db elements. 3. Write an <END CKPT> record to th log, and flush th log.

A transaction must not write any values (even to memory buffers) until it is certain not to abort.