CS186 MT1

- A database management system (DBMS) is software that stores, manages, and facilitates access to data.
- The most widely-used DBMS are relational, such as MySQL, Postgres, Oracle, etc.
- SQL (Structured query language) is the oldest and most common query language.

SQL

- Relations (tables) have a fixed schema, consisting of unique attribute names with atomic types. An instance is a multiset of rows/tuples.
- Attributes are synonomous w/ columns and fields; tuples are records or rows.

Data Definition Language (DDL):

```
CREATE TABLE Boats (
    bid INTEGER,
    bname CHAR(20),
    PRIMARY KEY (bid));

CREATE TABLE Reservations (
    sid INTEGER,
    bid INTEGER,
    day DATE,
    PRIMARY KEY (sid, bid, day),
    FOREIGN KEY (bid) REFERENCES Boats);
```

Data Manipulation Language (DML):

General queries are of the form

```
SELECT [DISTINCT] <column expression list>
FROM <table1 [AS alias1], ..., tableN [AS aliasN]>
[WHERE <predicate>]
[GROUP BY <column list>
    [HAVING <predicate>]]
[ORDER BY <column list>]      *ASC is default
[LIMIT <integer>]
```

Column expressions can be given aliases; in the SELECT clause, this controls the output field names.

Aggregate Queries

AVG, SUM, COUNT, MAX, MIN are examples of aggregates that can be used in the SELECT, HAVING, or ORDER BY clauses. GROUP BY partitions a table into groups with equal column values. Aggregates return only one row per partition.

Conceptual SQL Evaluation (not implementation!)

1) FROM identify tables, perform cross-product
2) WHERE eliminate rows
3) SELECT project away columns not used in SELECT, GROUP BY, HAVING
4) GROUP BY form groups and aggregate
5) HAVING eliminate groups
6) DISTINCT eliminate duplicates

Relational Algebra: operational description of computation.

Projection (π): retain only specified columns
Selection (σ): select subset of rows
Renaming: (ρ): rename output schema
Union ($\cup$): inputs must be type-compatible
Set-difference ($-$): in first but not 2nd
Cross-product ($\times$): combine relations

Intersect: $S_1 \cap S_2 = S_1 - (S_1 - S_2)$

Joins: $R \bowtie_\theta S = \sigma_\theta(R \times S)$

Equi-joins are theta-joins where θ is a conjunction of equalities. Natural joins are equi-joins on matching column names (so θ is implicit).

γ is the group by / aggregation operator.

- Nested queries:

```
WHERE ... [NOT] IN / [NOT] EXISTS
    <subquery>
```

checks whether the output tuple matches the results of the subquery.

Subqueries that reference a table in the outer query are called correlated and must be recomputed for every tuple.

- ANY/ALL are set comparison operators and are used w/ WHERE or HAVING clause. They return True if any/all of the subquery values meet the condition.

Examples: both return record(s) of youngest sailor(s):

- SELECT * FROM Sailors AS S WHERE s.age = ALL(SELECT MIN(age) FROM Sailors);
- SELECT * FROM Sailors AS S WHERE NOT EXISTS (SELECT * FROM Sailors AS S2 WHERE S.age > S2.age);

Other Joins We've been doing inner joins thus far. The following are equivalent:

```
... FROM Sailors AS S, Reservations AS R
    WHERE S.sid = R.sid ...
... FROM Sailors AS S INNER JOIN
    Reservations as R ON S.sid = R.sid ...
```

The full possibilities are

```
...FROM <table1> [INNER | NATURAL |
    {LEFT | RIGHT | FULL} OUTER] JOIN
    <table2> ON <qualifications> ...
```

- Inner joins are default. Natural joins are inner joins where the ON clause is implicity attributes w/ same name.
- Left outer join is same as inner join except unmatched rows from the left table are preserved (right-side fields are filled with NULLs). Right is same, but for right table, and full returns all unmatched rows on both sides.

Views: named queries (macros)

CREATE VIEW <name> AS <query>

can be referenced like normal tables; they are re-computed every time.

- A WITH clause makes a view on-the-fly by defining a table in advance.

```
WITH table-name(col1, col2, ...) AS (
    SELECT ...)
SELECT ...
```

- Can create many tables inside a WITH clause, just comma-separate them.

Ex: (argmax group by)

```
WITH maxratings(age, rating) AS (
    SELECT age, MAX(rating) FROM Sailors
    GROUP BY age)
SELECT S.* FROM Sailors as S, maxratings
    as m WHERE S.age = m.age AND
    S.rating = m.rating;
```

Miscellaneous SQL

- Null check with IS [NOT] NULL
- Null column values are ignored in aggregation, but aggregation over other columns (or all columns) will include those tuples.
- Combine queries with UNION [ALL], INTERSECT [ALL], EXCEPT [ALL].

Disks, Buffers, Files

- Note on memory: hierarchy of speed from registers → L1 cache → L2 cache → RAM → d
- SSDs have very fast reads compared to disk. Writes are slower due to limited # of erasures and write amplification. Random writes are 4x slower than sequential writes.
- SSDs are 10-100x faster than disk, but locality still matters and is far more expensive per unit capacity.

Files

Tables are stored as logical files consisting of pages of records.

Different DB file structures:

Unordered Heap files records placed arbitrarily across pages. Uses a header page connected to a linked list of full pages and a linked list of pages with free space. In a page directory design, there is a linked list of header pages, each with records giving # of free bytes on page & pointer to

Sorted Files pages and records are in sorted order of some column value.

Index Files may contain records or tell you where to look for record (e.g. B+ tree)

Page Layout Depends on record length (fixed / variable) and page packing (packed / unpacked).

For fixed-length records: Packed pages have header followed by densely packed records. Locating a record is arithmetic while deletion means repacking page. Unpacked pages are same, but use bit map. Deleting a record just means clear bit.

For variable-length records: slotted pages. Footer contains metadata with a 'slot directory'. The rightmost entry points to free space. For each other entry from right to left, the slot directory contains the length of the record and pointer to record.

To delete, null out slot directory entry.

To insert, place record in free space and create entry for it in next open slot.

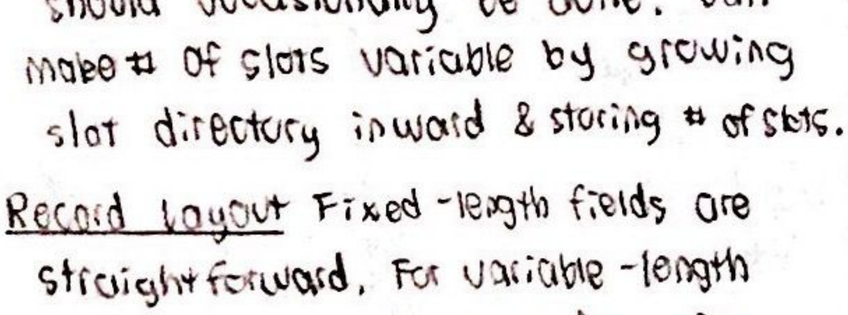
To avoid fragmentation, repacking should occasionally be done. Can make # of slots variable by growing slot directory inward & storing # of slots.

Record Layout Fixed-length fields are straightforward. For variable-length fields, introduce record header with pointers to the end of variable-length fields.

Indexes Heap files are good for full scans, frequent inserts/deletes. Sorted files are good for equality search and range search. But what about quick lookup by value? Enter indexes.

An index is a disk-based data structure that enables fast lookup and modification of data by search key (subset of columns). We'll assume data entries are (key, record id) so pointers to records in heap files.

B+Trees common indexes



Occupancy invariant: $d \leq$ # of entries $\leq 2d$

where d is the order of the tree

max-fan-out $= 2d+1$

The number of records in a maximally full B+tree is $(2d+1)^h \times 2d$, where h excludes the root.

Insert algorithm:

1) Find correct leaf L.
2) Put data in L in sorted order. If enough space, done. Else, split L into two nodes (with middle key going right). Copy up middle key to L's parent and add pointer to right node.

The copy-up step can be recursive. To split an interior node, push (not copy) the middle key up.

In practice, deletion often just removes the leaf entry and doesn't re-balance. So occupancy-invariant doesn't hold.

Bulk-loading: efficient construction of B+tree. First sort input records, then start filling leaf nodes up to some occupancy factor (e.g. 3/4) with continuous splitting as necessary. The lower left of the tree is not touched again, leading to good cache locality. during construction.

- An index is ordered by a column (search key) or multiple columns (composite key). A composite search key on columns $k_1, k_2, \ldots, k_n$ "matches" a query if: the query is a conjunction of $m \geq 0$ equality clauses: $k_1 = v_1$ AND ... AND $k_m = v_m$. and at most 1 additional range clause of the form AND k_{m+1} op val where op is $\{<, >, \leq, \geq\}$.
- 3 alternatives for data representation in an index:
  1) By value - store entire record contents
  2) By reference - store pointer to data record
  3) By reference list - pointers to records sharing the same key

We use 'by reference' representations to allow multiple indexes per DB.

- In a 'clustered' index, the underlying heap records are kept mostly ordered by search keys in index.

Hashing and sorting have same memory and I/O cost for 2 passes (which works for any file with $\leq B(B-1)$ pages). Hashing is good for duplicate elimination and is easily parallelizable. Sorting scales w/ number of duplicates and can suffer from 'data skew' when trying to produce cutoff ranges for partioning data to be sorted in parallel.

Cost Summary

	Heap file	Sorted file	Index*
Scan all records	BD	BD	3/2 BD
Equality	0.5BD	$\log_2(B)\,D$	$(\log_F(BR/E)+1)D$
Range	BD	$(\log_2(B)+\text{pages})D$	$(\log_F\frac{BR}{E}+\text{pages})D$
Insert	2D	$(\log_2(B)+B)D$	$(\log_F(BR/E)+3)D$
Delete	0.5BD + D	$(\log_2(B)+B)D$	$(\log_F(BR/E)+3)D$

*We assume index is clustered with pages 2/3 full.

B = # of data blocks
R = # of records per block
D = avg time for I/O on block
F = average internal node fanout
E = average # of data entries per leaf

Buffer Management

The DB buffer manager stores map of frame id, page id, dirty bit, and pin count. Replacement involves choosing unpinned page, writing back, reading new page into frame, pinning the page, and returning the address. The requestor is responsible for unpinning a page; the manager must stall if all pinned.

Page replacement policies

LRU: each frame stores time it was last unpinned.

Clock: faster approximation of LRU.

Algo: If page in buffer, set ref bit. Else, check frame of current clock hand. If ref bit set, clear and advance hand. Repeat until ref bit = 0. Then replace page, set ref bit, and advance clock hand.

LRU and clock suffer from sequential flooding. MRU actually works better for this scenario.

Out-of-core algorithms

External Merge Sort: Sort a file w/ N pages using only B buffer pages.

Pass 0: Use all B pages to produce $\lceil \frac{N}{B} \rceil$ runs of length B pages each. (i.e read as much data as possible, run quicksort and repeat until entire file processed)

Pass 1, ..., n: merge B-1 runs at a time, using last buffer as output. Each one of these passes divides # of runs by B-1 and multiplies length of run by B-1.

Number of passes: $1 + \lceil \log_{B-1}(\lceil \frac{N}{B} \rceil) \rceil$

I/O cost is $2N *$ number of passes.

External Hashing: make matching records consec

Phase 1: Divide. Using single input buffer, hash each record to one of B-1 output buffers.

Phase 2: Conquer. For each of the B-1 hashed partitions, use all B buffer pages on hash table that we write to disk.

I/O cost is $4N +$ (2 × # of pages needing recursive partitioning)

CS186 Midterm 2

Iterators

- A logical query plan can be visualized as a graph where vertices are relational algebra operators and edges encode data flow.
- Each operator implements an iterator interface - setup(inputs), init(args), next(), close().
- A query plan is executed by calling init() and next() on the root vertex. These calls create recursive init and next calls down the operator chain; some will be streaming and some will be blocking.

Join Operators For relation R, let [R] denote # of pages to store R; pR denote the # of records per page of R; |R| denote the cardinality of R ([R] × pR).
Let R be on the left (outer loop) and S be on the right (inner loop) for a join.

Simple Nested Loops I/O cost: [R] + |R| * [S]

Page-nested Loop Join I/O cost: [R] + ([R] * [S])

Block-nested Loop Join [R] + ⌈[R]/(B-2)⌉ * [S]

psuedocode:

```
for each rblock of B-2 pages of R:
  for each spage of S:
    for all matching tuples in spage and rblock:
      yield <rtuple, stuple>
```

This is the best we can do for non-equi joins.

Index-Nested Loops Join

Unclustered: [R] + |R| * (Search + HeapAccess)
Clustered: [R] + |R| * (Search) + distinctvals(R) * heapAccess

Search is usually height of B+ tree +1, and heapAccess is usually 1.

Sort-Merge Join combines sorting + joining on an attribute (equality predicate)

2 stages: 1) Sort R and S by join key (externally)
2) Join Pass: Merge-scan sorted partitions

The join pass moves two pointers, r and s, until the values match. Stream down s, emitting tuples until no longer match. Reset s, increment r, and repeat.

Cost: Sort R + Sort S + ([R] + [S])

In the worst case (rare), last term becomes ([R] * [S]).

Optimization: If $B \geq \sqrt{R} + \sqrt{S}$, we can sort both relations in memory simultaneously and do join during final merge pass of sort. saves 1 pass each, or [R]+[S] I/Os.

Grace-Hash Join (equality predicate)

Partition R and S by hashing on join key and write to disk. Load pairs of partitions into memory, build a hash table on the smaller partition, and probe the other for matches with the current hash table.

Cost:

Partitioning phase: 2([R] + [S])
Matching Phase: [R] + [S]
Total cost: 3([R] + [S])

This assumes [R] < (B-1)(B-2). If not, need to recursively partition smaller relation:

([R] + [S])(1 + 2 * # of partion phases)

Naive hash join is Grace Hash without partioning phase. If R fits in memory to build the hash table, cost is just [R]+[S].

Sorting vs Hashing (for joins)

Sorting is good if one or both relations are already sorted. It is not sensitive to data skew. Hashing is good if one of the relations is much smaller.

Parallel Query Processing

2 main kinds of Parallelism:

Pipeline: Each stage is doing something on the result of the previous stage. Scales up to depth of pipeline.

Partition: Split up data and process at the same time. Scales up to amount of data.

Parallel Architectures

Shared memory: several CPUs sharing RAM and disk, e.g. multicore computers.

Shared disk: separate RAM, same disk.

Shared nothing: only pass data over network e.g. clusters. common in practice.

Interquery parallelism: each query runs on a separate processor.

Intraquery parallelism: parallelism within single query, e.g. pipeline parallelism.

Intra-operator parallelism (via partitioning)

Different partitioning schemes...

Range: assign tuples to partitions based on column values falling within given range. Good for range queries, equijoins, group by.

Hash: hash column value to a partition. Good for equijoins, group-by.

Round-robin: assign tuples to partitions in sequence. Good for spreading load.

Scans, lookups, and inserts can be performed in parallel using appropriate broadcasts, shared output streams, etc.

Parallel Grace-Hash Join

Pass 1: Shuffle data across machines with hash h_m. Build table and then stream the probing relation through it.
Pass 2: As probe data streams in, proceed with local Grace-Hash join.

Near perfect speed-up, only waiting is for pass 1 to finish.

Parallel Sort-Merge Join

1) Partition both relations into equal-sized partitions (random sampling can guide this)
2) Perform parallel external merge sort.
3) Using refinement, merge-join partitions locally on each node in final merge pass.

The number of passes (reading over 1 relation)
$= 2 + (1 + \log_{B-1}(\frac{[R]}{mB})) + (1 + \log_{B-1}(\frac{[S]}{mB})) + 2$

(cost of sorting + cost of partioning + merging)

Parallel aggregates are straightforward; need local/global decomposition. e.g. average needs local sums and counts.

Symmetric (pipelined) hash join

Grace-Hash is a "pipeline-breaker" because probing can't start until hashtable is built. If there is enough memory. can have each node build 2 hashtables; upon arrival of R tuple, build it into R hash table and probe S hashtable for matches (and vice-versa for S tuple).

Note we've looked at symmetric shuffle joins. In asymmetric shuffle, one of the tables is already partioned on join key. In broadcast join, if one relation is small, broadcast it to all nodes containing larger relation and do local merge+join.

Query Optimization

A query first goes to a query parser, which checks correctness, authorization, and generates a parse tree. Then the query rewriter converts queries to canonical form (removes views, subqueries, etc.). The query optimizer optimizes one query block at a time, using catalog stats to find least (estimated) cost plan per block.

Relational Algebra Equivalences

Selections: (cascade)
$\sigma_{c_1 \wedge \ldots \wedge c_n}(R) \equiv \sigma_{c_1}(\ldots(\sigma_{c_n}(R))\ldots)$
$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$ ← commutativity

Projections: (cascade)
$\pi_{a_1}(R) \equiv \pi_{a_1}(\ldots(\pi_{a_1,\ldots,a_{n-1}}(R))\ldots)$

Cartesian Products:
$R \times (S \times T) \equiv (R \times S) \times T$ ← associative
$R \times S \equiv S \times R$ ← commutative

Common Heuristics

- Selection pushdown & cascade
  Apply selections as soon as we have relevant columns
- Projection pushdown & cascade
  Keep only columns needed to evaluate downstream operators.

- Avoid Cartesian products
  Always choose θ-joins rather than cross-products (usually good bet)

Physical Equivalences are equivalent algorithms. Ex: heap scan vs index scan. For equijoins, can use block nested loop, index nested loop (if possible), sort-merge join (good for small, equal-sized tables), or Grace hash join (good if 1 table small). For non-equijoins, only choice is BNL.

Plan space
Even for simple query, many possibilities
- applying selection push-down
- reversing join order
- materializing result of selection on inner loop to avoid rescanning table
- using sort-merge join over BNL.
- applying projection pushdown
- Use index scan over full scan

System R (Selinger) optimizers work by considering only left-deep plans and avoiding Cartesian products. Not always best idea, but ok heuristic. For each query block, consider
- All relevant access methods
- All left-deep join trees; right branch always a base table

A full plan space includes all logical equivalences as well as physical properties of implementation (e.g. sorted/grouped order).

Cost Estimation (We only consider I/O costs)
Must estimate (1) cost of operator; and (2) size of result, which determines downstream operator costs.

Selectivity = |output| / |input|
Assuming evenly distributed data, Selinger uses the following estimations:

Term	selectivity
col = val	1/Nkeys(C)
col1 = col2	1/MAX(Nkeys(C1), Nkeys(C2))
col > val	(MAX(C) - val) / (MAX(C) - MIN(C) + 1)

If stats are unknown, we assume sel = $\frac{1}{10}$
The result cardinality is then input tuples × product of selectivities.

Join selectivity is selectivity of predicate times |R| × |S|.

Histograms
We can get better estimates by storing histograms of column values.
For example, estimating selectivity of T.a = T.b using equi-width histogram:

```
s=0
for values v covered in both histograms:
    s += height(bina(v))/(nr*width(bina(v)))
         *height(binp(v))/(nr*width(binp(v)))
```

Search Algorithms

Single-relation plans consider each access path and choose one with least estimated cost, considering selection & projection.

#of pages retrieved: (WHEREs in SQL)

Clustered Index I on 1 or more selects:
(NPages(I) + NPages(R)) * prod of selectivities of matching selects

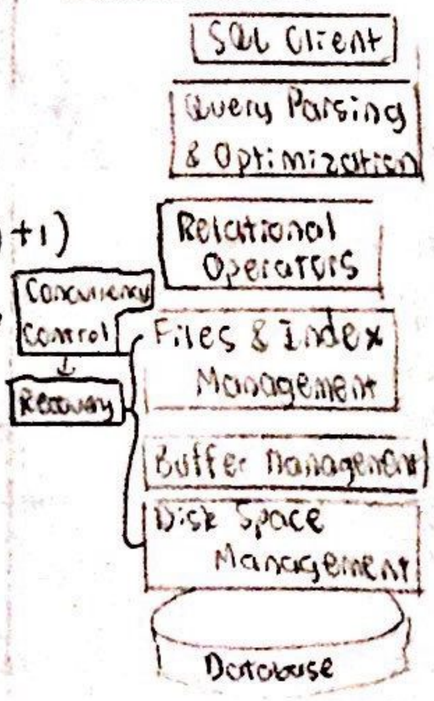
Unclustered Index I on 1 or more selects:
(NPages(I) + NTuples(R)) * prod of selectivities of matching selects

Sequential File Scan
NPages(R)

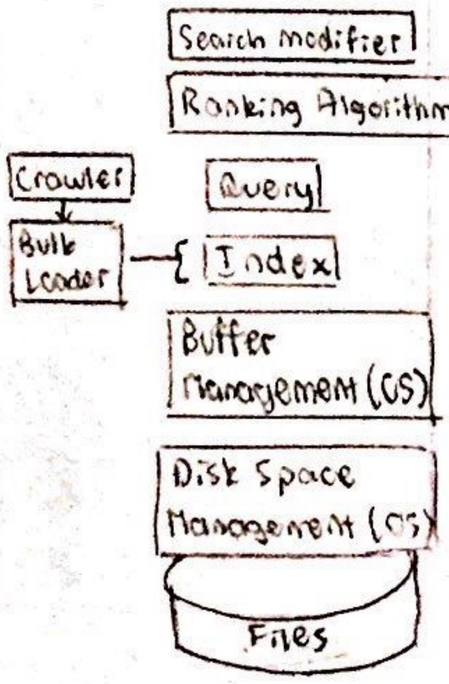
Dynamic Programming Algorithm for building query plan:
Pass 1: Find best plans of height 1 (base table accesses) and record in table.
Pass i: Find best plan of height i by combining plans of height i-1.

An intermediate result has an "interesting order" if it is sorted by anything useful downstream in the query.
The DP table records, for each subset of joined tables thus far and interesting order columns, the best plan and its cost.
The algorithm gives the types of scans and joins to perform. ORDER BY, GROUP BY, aggregates are done as a post-processing step (by additional sort/hash operator if needed).
This algorithm is factorial in the number of tables.

Text Search (Information Retrieval Intro)



DBMS Architecture



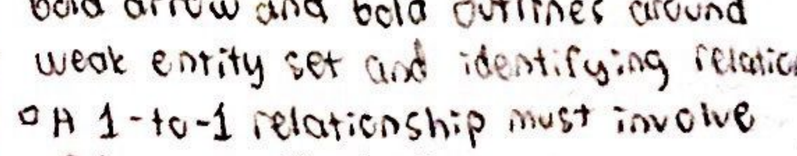
IR Architecture

- Key difference is updates occur infrequently with bulk loads from (web) crawler.
- Typical IR data model is "bag of words"; each document is a bag of terms.

Boolean Text Search Return all documents matching Boolean expression, e.g "A" AND ("B" OR "C") AND NOT "D".

- A corpus (collection) of text files consists of files, which are (docID, content text) pairs. We can construct an InvertedFile, or mapping of terms to DocIDs.
- We build an alt. 3 index on InvertedFile, keeping the list of docIDs ("postings list") in sorted order.
- To handle ORs, union the term lookup results. For ANDs, intersect the results. Handle AND NOTs with set subtraction.
- All queries are just a series of merge joins on index scans.
- Handle phrases by augmenting InvertedFile with int position and unioning results that are 1 position off.
- In general, text search engines are an example of engineering for very special case workload (a common theme in DBs).

Entity-Relationship Models



Entities (rectangles) are real-world things with attributes (ovals) and a primary key (underlined). A relationship (diamond) is an association of entities, which may also have attributes.

Key Constraints Arrow from entity to relationship to represent that source entity participates at most once (a 1-to-many relationship).

Participation Constraint Bold edge to represent all members of entity set participate (full participation) or at least one (partial participation).
⇒ Bold arrow means 'exactly 1'.

A weak entity is a dependent entity on an owner entity, in a 1-to-many relationship. It has full participation in the identifying relationship. Drawn with bold arrow and bold outlines around weak entity set and identifying relationship.

- A 1-to-1 relationship must involve 2 key constraints.
- A dotted box around a subgraph denotes aggregation, or encapsulated entity set.
- ER models are meant to capture semantics of real world, then be translated to relational models.

CS186 Final

Functional Dependencies are a form of integrity constraints that identify redundancy in schemas and help suggest refinement.

- A FD $X \to Y$ ("X determines Y") holds if given any two tuples in table R with the same X values, their Y values are the same. (X & Y are attribute sets).

Superkey (or just key): Set of columns that determines all columns in its table.

Candidate key: minimal set of columns that determines all columns in its table.

A primary key is simply a chosen candidate key.

Armstrong's axioms Denote F+ as the closure of F, or set of all FDs that are implied by F (including trivial dependencies). Let X, Y, Z be sets of attributes:

Reflexivity: If $X \supseteq Y$, $X \to Y$ (if X is superset of Y, $X \to Y$)

Augmentation: If $X \to Y$, then $XZ \to YZ$ for any Z

Transitivity: If $X \to Y$ and $Y \to Z$, then $X \to Z$.

These rules are complete: can derive all of F+ using them. Some logical consequences:

Union: If $X \to Y$ and $X \to Z$, then $X \to YZ$.

Decomposition: If $X \to YZ$, then $X \to Y$ and $X \to Z$.

To check if $X \to Y$ is in F+:

- Compute attribute closure of X (denoted X+) w.r.t. F, which is the set of all attributes A s.t. $X \to A$ is in F+:

```
X+ := {X}
Repeat until no change:
    for U→V in F:
        if U ⊆ X+, add V to X+
```
(for each FD in F, if U is in X+, add V to X+)

- Then check if Y is in X+.

If X+ = R, then X is a superkey of R. Further, it is a candidate key if for all columns A in X, $(X-A)+ \neq R$.

BCNF Normal Form A relation R with FDs F is in BCNF if, for all $X \to A$ in F+:
$A \subseteq X$ (a trivial FD), or X is a superkey of R. If this is the case, every field holds useful info that cannot be inferred from F.

- Potential issues converting R into BCNF:
  1) Impossible to reconstruct original relation (lossiness)
  2) Enforcing FDs (dependency checking) may require joins
  3) Queries become more expensive.

1) is correctness property, 2)-3) are performance.

Theorem: The decomposition of R into X and Y is lossless w.r.t. F iff F+ contains
- $X \cap Y \to X$, or $X \cap Y \to Y$.

In other words, the shared column(s) of the decomposition must be superkeys of the first or second relation.

- If R is decomposed into X and Y, the projection of F on X (F_X) is the set of FDs $U \to V$ in F+ s.t. all attributes U, V are in X.
- The decomposition of R into X and Y is dependency-preserving if $(F_X \cup F_Y)+ = F+$.

Converting to BCNF: If $X \to Y$ violates BCNF, decompose R into R-Y and XY. Repeating this gives collection of relations in BCNF. Guaranteed to be lossless.

- In general, BCNF can always be done losslessly, but there may not be a dependency preserving decomposition into BCNF.
- BCNF is a good heuristic - other normal forms are more permissive and can be better.

Transactions & Concurrency

Advantages of concurrent execution: improved throughput (transactions per second) and improved latency (response time per transaction).

A transaction is a sequence of multiple actions to be executed as an atomic unit.

High-level properties of transactions (ACID):

Atomicity: all or no actions occur
Consistency: DB remains consistent after Xact
Isolation: execution doesn't affect others ← criterion: serializability
Durability: effects persist

Concurrency Control To the Xact manager, a Xact consists only of Begin, Read/Writes, Commit, or Abort.

- Serial schedule: Each transaction runs from start to finish without intervening actions from others.
- Schedules are equivalent if they involve the same transactions; each individual transaction's actions are ordered the same; both schedules leave the DB in the same final state.
- A schedule is serializable if it is equivalent to some serial schedule.
- Two operations conflict if they are by different transactions; are on the same object; and at least one of them is a write.
- Two schedules are conflict equivalent iff they involve the same actions of the same transactions, and every pair of conflicting actions is ordered the same way.
- A schedule is conflict serializable if it is conflict equivalent to some serial schedule. In other words, S is conflict serializable if you can transform S into a serial schedule by swapping consecutive non-conflicting operations.
- All conflict serializable schedules are serializable, but not vice-versa. Hence it is a conservative test.

Conflict Dependency Graph: one node per transaction, edge from $T_i \to T_j$ if an operation o_i of T_i conflicts with an operation o_j of T_j AND o_i appears earlier in schedule than o_j.

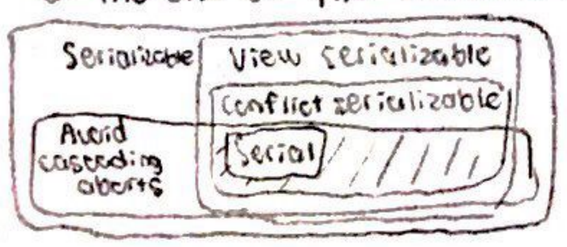
- A schedule is conflict serializable iff its dependency graph is acyclic.
- Another definition of serializability is view serializability, which allows "blind writes": non-final writes that aren't preceded by reads.

Scheme for enforcing conflict serializability:

2PL (Two-Phase Locking)
- A Xact must obtain a S lock before reading and an X lock before writing.
- Xact cannot get new locks after releasing any lock.

2PL guarantees conflict serializability, but not cascading aborts/rollbacks.

Strict 2PL prevents cascading aborts by requiring all locks to be released together at the end of the transaction.



Venn diagram of schedules

- The lock manager maintains a hashtable keyed on resource names. Each currently held lock is an entry with a granted set: set of Xacts granted access to lock; lock mode: type of lock held; and a wait queue of lock requests.
- When a lock request arrives: does any xact in granted set or wait queue want a conflicting lock? If no, put requester in granted set and proceed. Else, put requester in wait queue.
- Lock upgrade: To avoid deadlock, put request in front of waiting queue.
- Dealing with deadlock:
  1) Prevention: Whereas the OS can specify a resource ordering, we can't do this in databases.
  2) Avoidance: assign priorities based on age (now - start time). Say T_i wants a lock T_j holds. Two possible policies:
     Wait-Die: If T_i priority higher, T_i waits for T_j, otherwise T_i aborts.
     Wound-wait: If T_i priority higher, T_j aborts, otherwise T_i waits.
  3) Detection & Resolution: Create & maintain "waits for" graph: $T_1 \to T_2$ if T_1 waits on T_2. Note that edge goes to last Xact on wait queue. Periodically check for cycles and kill a transaction in the cycle.

Multigranularity Locking

Allows locks on DB → tables → pages → records.

	IS	IX	S	SIX	X
IS	Y	Y	Y	Y	N
IX	Y	Y	N	N	N
S	Y	N	Y	N	N
SIX	Y	N	N	N	N
X	N	N	N	N	N

To Get:	Parent must hold:
IS or S	IS or IX
IX or SIX or X	IX or SIX

Recovery

In SQL, transactions are handled via BEGIN; COMMIT; ROLLBACK <7 statements.

There are also savepoints:

SAVEPOINT <name>: create savepoint

RELEASE SAVEPOINT <name>: remove savepoint

ROLLBACK TO SAVEPOINT <name>: statements since savepoint are rolled back

Our preffered recovery policy is NO FORCE/STEAL, which requires UNDO & REDO logging.

A log is an ordered list of log records, each of which contains <XID, pageID, offset, length, old data, new data>.

Write-Ahead Logging Protocol (WAL):

1) Force the log record for an update before the corresponding data page gets to the DB disk.
2) Force all log records for a Xact before commit.

1) helps guarantee atomicity & 2) helps guarantee durability.

- The log is stored as an ordered file with a write buffer ("tail") in memory. The tail is periodically flushed to disk. Each log record has a log sequence number (LSN), which is unique and increasing.
- The largest LSN written to disk so far ("flushedLSN") is also tracked in RAM.
- Each data page in the DB contains a pageLSN, a pointer to the most recent log record for an update to that page.

WAL protocol enforces:

Before page i is written to DB, log must satisfy pageLSN_i ≤ flushedLSN.

Aries Log Records

Log Record fields: LSN, prevLSN (LSN of previous log record by XID), XID, type (one of update, commit, abort, checkpoint, compensation log record (CLR), or END). Update records additionally include pageID, length, offset, before-image, after-image.

Aries maintains state in memory:

Transaction Table: one entry per currently active Xact, containing XID, status, lastLSN (most recent LSN written by Xact).

Dirty Page Table: one entry per dirty page in buffer pool, containing recLSN (LSN of log record which first caused page to be dirty).

- At commit time, write commit record to log. Once all log records flushed to disk, the commit can return. Write end record to log for bookkeeping.
- CLRs are used for undoing actions: for each undone action, the "undonextLSN" field points to the next LSN to undo (the prevLSN of undone record). Undo is not idempotent.
- To make recovery in event of crash faster, use checkpointing: snapshot of current in-memory state tables.

Write to log: begin_checkpoint record. Then end_checkpoint record containing current Xact table and DPT. The LSN of the most recent checkpoint is stored as the master record, often first block of log file.

ARIES crash recovery

Phase 1: Analysis - figure out state since checkpoint. Starting from DPT and Xact table from checkpoint, scan log forward from begin checkpoint record. If we encounter...

End record: remove Xact from Xact table.

Update record: add P to DPT and set its recLSN = LSN (if not already in DPT)

Any other record: add Xact to Xact table, set lastLSN = LSN, and change Xact status on commit.

At end of analysis, for any Xacts in Xact table in commiting state: write END log record and remove from table.

Phase 2: Redo - reapply all updates & redo CLRs.

For each update/CLR, redo unless

- affected page not in DPT;
- affected page in DPT, but recLSN > LSN;
- OR pageLSN ≥ LSN

Otherwise redo action & set pageLSN = LSN.

Phase 3: Undo - undo effects of failed Xacts.

```
toUndo = {lastLSNs of all xacts in xact table}
while !toUndo.isEmpty():
    thisLR = toUndo.remove-largest-LSN()
    if thisLR.type == CLR:
        if thisLR.undoNextLSN != null:
            toUndo.insert(thisLR.undoNextLSN)
        else:
            write end record for xid to log
    else: # is an update
        write CLR for the undo
        undo update in DB
        toUndo.insert(thisLR.prevLSN)
```

- If system crashes during analysis: nothing serious, just start over next time.
- If system crashes during Redo: we'll detect any Redo's done already and not do them again.
- To limit work in Redo, flush asynchronously so that oldest recLSN isn't too old.

Search Ranking

Canonical scheme is TD × IDF

```
for each doc d:
    docTermRank = # occurrences of term in d ×
        log(total # docs / # docs with this term)
```

Represent documents as vectors whose i^{th} entry is the docTermRank of term i.

Measure similarity with cosine similarity (just dot product after normalizing).

Precision: % of output that is correct

Recall: % of correct results that are output

Distributed Transactions w/ 2Phase Commit

Phase 1

- Coordinator tells participants to prepare
- Participants generate prepare/abort record for the log
- Participants flush prepare/abort record
- Participants respond with yes/no votes
- Coordinator generates commit record for log
- Coordinator flushes commit record

Phase 2

- Coordinator broadcasts result of vote
- Participants make commit/abort record
- Participants flush commit/abort record
- Participants respond with Ack
- Coordinator generates END record
- Coordinator flushes END record

If coordinator sends COMMIT at any time, it must commit.

Recovery: If participant is down: if participant hasn't voted yet, abort. If waiting for Ack, hand to "recovery process". If coordinator is down: if it hasn't logged prepare, abort. If it has logged prepare, hand to "recovery process."

"Recovery process": coordinator gets inquiry from "prepared" participant. If Xact table at coordinator says aborting/committing: send appropriate response & continue. If coordinator says nothing: inquirer aborts.

What happens when coordinator recovers?

- with commit and end: nothing
- with just commit: rerun phase 2
- with abort: nothing

What happens when participant recovers?

- with no prepare/commit/abort: nothing
- with prepare & commit: send Ack
- with prepare: send inquiry
- with abort: nothing

2 Phase Commit is not great for availability, as recovery needs the coordinator to recover before moving on.