CO572 Advanced Databases

1 Database management systems

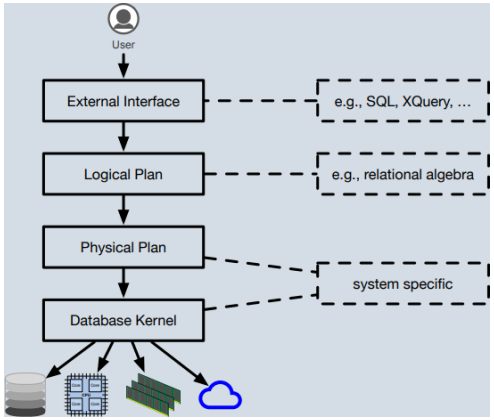
Requirements Should provide:

- *Storage*: single, reliable repository of data.
- *Transactions*: should be atomic, consistent, isolated, durable.
- *Data analysis*.
- *Programming model*.

In a way that is:

- *Efficient*: not slower than hand-written applications.
- *Resilient*: should recover from problems.
- *Robust*: predictable performance.
- *Scalable*: use resources efficiently.
- *Concurrent*: serve multiple simultaneous clients transparently.

Internals



Usage Models

- **OLTP:** online transaction processing: lots of small updates - want high throughput, ACID guarantees.
- **OLAP:** online analytical processing: running a single data analysis task - want low latency, queries are ad-hoc.
- **Reporting:** running many data analysis tasks in a fixed time - want good resource efficiency, queries known in advance.
- **HTAP:** hybrid transactional/analytical processing.

2 Storage

Storage manager Needs to implement (at least):

1. Insert tuple.
2. Delete tuple by key.
3. Find tuple by key.

N-ary storage model

- Simply store full tuples in a vector.
- Insertion is as simple (append the tuple).
- Data locality is bad when retrieving.

13 Temporal databses

Temporal dimenstions

1. *Valid time*: time data valid in Universe of Discourse.
2. *Transaction time*: time data valid in DBMS.

Modelling a flow of time

1. *Discrete* vs *continuous*.
2. *Unbounded* vs *bounded*.
3. *Linear* vs *branching*.

US-Logic

1. A *until* B: A holds at every time up to an including the time when B holds.
2. A *since* B: A has held at every time since and including the time when B held.
3. A *since product* B: $\text{eval}\left(A \overset{S}{\times} B, t\right) = \text{eval}\left((A \times B) \cup \left(\left(A \overset{S}{\times} B\right) \ltimes A\right), t-1\right)$.
4. A *until product* B: $\text{eval}\left(A \overset{U}{\times} B, t\right) = \text{eval}\left((A \times B) \cup \left(\left(A \overset{U}{\times} B\right) \ltimes A\right), t+1\right)$.

We can derive:

1. A *since join* B: $\text{eval}\left(A \overset{S}{\bowtie} B, t\right) = \text{eval}\left(\sigma_{A.x=B.x}\left(A \overset{S}{\times} B\right), t\right)$.
2. A *until join* B: $\text{eval}\left(A \overset{U}{\bowtie} B, t\right) = \text{eval}\left(\sigma_{A.x=B.x}\left(A \overset{U}{\times} B\right), t\right)$.
3. $\bigcirc A \equiv A$ Until $\top$: A is true at the next time.
4. $\bullet A \equiv A$ Since $\top$: A is true at the previous time.
5. $\Diamond A \equiv \top$ Until A: A is true at some time in the future.
6. $\blacklozenge A \equiv \top$ Since B: A is true at some time in the past.
7. $\Box A \equiv A$ Until $\neg\bigcirc\top$: A is always true in the future.
8. $\blacksquare A \equiv A$ Since $\neg\bullet\top$: A is always true in the past.

Strong strict locking All locks released at end of transactions.

- + Simple to implement.
- + Suitable for distributed transactions.

SQL transaction isolation level `SET TRANSACTION ISOLATION LEVEL ...`

1. `READ UNCOMMITTED` prevents only dirty writes.
2. `READ COMMITTED` prevents dirty reads and writes.
3. `SNAPSHOT` prevents all anomolies except write skew.
4. `REPEATABLE READ` prevents all anomolies except phantom reads.
5. `SERIALIZABLE` implements full serialisability.

Distributed concurrency control *DWFG*:

1. Add spawn double-edges to master.
2. When a local cycle appears, fetch remote WFG.

Global 2PL: 2PL cannot just be executed at each site:

1. Use strong strict locking at each site, using a global atomic commit to end transaction.
2. *Two-phase commit*:
   - Can become blocked.

service element	source	semantics
C-PREPARE	coordinator	get ready to commit
C-READY	server	ready to commit
C-REFUSE	server	not ready to commit
C-COMMIT	coordinator	commit the transaction
C-ROLLBACK	server	rollback the transaction
C-RESTART	either	try to return to start of transaction

Distributed locking: for n sites,

- *Write-write conflicts*: at least $\lceil \frac{n+1}{2} \rceil$ sites must be sent write lock.
- *Read-write conflicts*: at least $n - k + 1$ sites must be sent a read lock, where k is the number of write locks.

Decomposed storage model

- Each column stored in a separate vector.
- Insertion now requires decomposition (more complicated, requires random access).
- Retrieving a single tuple requires reconstructing.

Metadata

- If the table is *dense* (all keys are consecutive), we can have constant lookups.
- If the table is *sorted*, we can use binary search.

Hybrid storage: Delta/Main

- Main storage uses DSM.
- Delta storage uses NSM.
- Periodically merge delta into main.

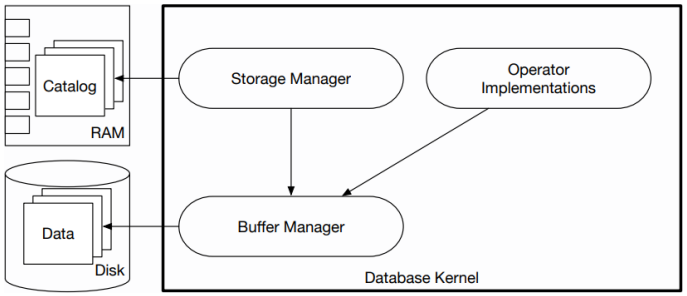
Variable-sized datatypes

1. *In-place storage*: overallocate space.
   - Good for locality.
   - Simple.
   - Wastes space.
2. *Out of place storage*: key into a dictionary.
   - Space conservative (we can even do dictionary compression).
   - Bad for locality.
   - Complicated (hard to implement: garbage-collection).

Disks

1. Larger pages.
2. Much higher latency (ms vs ns).
3. Much lower throughput.
4. OS gets in the way (filesize is limited).

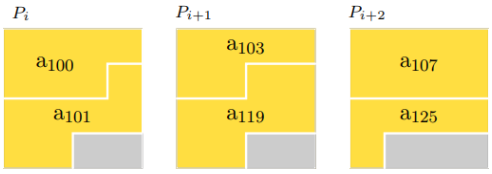
Change of goals:



1. Complicated I/O management strategies pay off.
2. Pages are large: we use a *buffer manager* to manage pages for a relation.

Pages

1. *Unspanned pages*:



   - Simple.
   - Assuming known page fill-factors, we can always lookup an ID with a single page lookup.
   - Wastes space.
   - Cannot deal with large records.
   - No in-page random access for variable sized records.

2PL Protocol Every history has a maximum lock period, we must be able to re-time history to make all operations take place during that period $\implies$ CSR.

1. *Read locks* (non-exclusive).
2. *Write locks* (exclusive).
3. *Growing* and *shrinking* phase: cannot gain a new lock after releasing a lock.

Anomaly	Pattern	Prevented by
Dirty Write	$w_1[o] \prec w_2[o],\ w_2[o] \prec e_1$	Strict 2PL
Dirty Read	$w_1[o] \prec r_2[o],\ r_2[o] \prec e_1$	Strict 2PL
Inconsistent Analysis	$r_1[o_a] \prec w_2[o_a],\ w_2[o_b] \prec r_1[o_b]$	2PL
Lost Update	$r_1[o] \prec w_2[o],\ w_2[o] \prec w_1[o]$	2PL
Simple Write Skew	$r_1[o_a] \prec w_2[o_b],\ r_1[o_b] \prec w_2[o_a]$	2PL
Write Skew	$r_1[P_1] \prec w_2[x \in P_2],\ r_2[P_2] \prec w_1[y \in P_1]$	2PL with Predicate Locks
Phantom Read	$r_1[P_1] \prec w_2[x \in P_1],\ w_2[y \in P_2] \prec r_1[P_2]$	2PL with Predicate Locks

2PL *does not prevent phantom reads / complex write skews*. Solutions:

1. *Table locks*: read lock table when performing a scan:
   - $-$ Can produce needless conflicts.
   - $+$ Efficient if large parts of the table are being updated.
2. *Predicate locking*: lock the predicate that the transaction uses.
   - $-$ Difficult to implement.

We can get *deadlocks* with 2PL: if WFG has a cycle.

Conservative locking Obtain all locks at beginnning.

- $+$ Prevents deadlock.
- $-$ Quite difficult to tell what locks required.
- $-$ Not recoverable.

Strict locking Prevent write locks being released before end of transaction.

- $+$ Prevents dirty reads/writes $\implies$ recoverability.
- $-$ Allows deadlock.

5. *Phantom read*: T_1 executes two queries, the second returning something different to the first, due T_2 inserting/removing an object.
6. *Write skew*: T_1 and T_2 concurrently read overlapping data and then concurrently make updates.

Conflicts An interaction between two transactions:

1. $r_x[o]$ and $w_y[o]$ are in H, or
2. $w_x[o]$ and $w_y[o]$ are in H.

H_1 and H_2 are conflict equivalent if:

1. They contain the same set of operations, and
2. Order conflicts (of non-aborted transactions) in the same way.

H is conflict serialisable if $C(H) \equiv_{CE} H'$ where H' is a serial history.

Serialisation graph Contains a node for each transaction in H, and an edge if there as a conflict from one node to another.

- If $SG(H)$ is acyclic, then H is conflict serialisable.

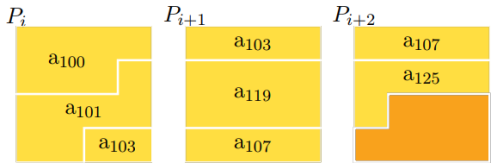
Recoverability In addition to recoverability, we sometimes want to ensure:

- *Avoids cascading aborts*: no dirty reads.
- *Strict execution*: no dirty reads or writes.

Maintaining serialisability and recoverability

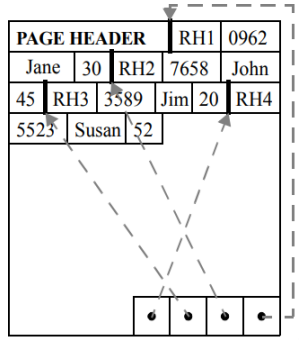
1. *Two-phase locking*: conflict based.
2. *Time-stamping*: add timestamps to object, only read or write objects with earlier timestamp.
3. *Optimistic concurrency control*: inspect history for problems at commit.

2. *Spanned pages*:



- Minimises space waste.
- Supports large records.
- Complicated.
- Random access performance worsened.
- No in-page random access for variable-sized records.

3. *Slotted pages*:



- Store tuples in in-place n-ary format.
- Store tuple count in page header.
- Store offsets to every tuple. Only need to be large enough to address page.

Dictionaries usually kept in-page:

- Solves problem of variable sized records.
- Avoids needless page faults.
- Allows duplicate elimination.

3 Querying

1. *Vector*: ordered collection of objects of the same type.
2. *Tuple*: unordered collection of objects of different type.
3. *Bag*: unordered collection of objects of the same type.
4. *Set*: unordered collection of unique objects of the same type.

Relational algebra $\pi_{\text{attrs}}(A)$, $\sigma_{\text{pred}}(A)$, $A \times B$, $A \cup B$, $\Gamma_{\text{(attrs, (attr, aggregate, name))}}$, $\tau_{\text{(n, attr)}}$.

4 Processing models

Volcano processing

- Easy to implement: focus on flexibility, clean design, maintainability, developer productivity.
- *Extensible*: easy to add new operators.
- *Good I/O behaviour*: tuples consumed as they are produced.
- *Poor CPU usage*: function pointers cause control hazards.
- Uses `open()`, `next()` and `close()` methods.

1. *Scan*:
   - `open()`: open file.
   - `next()`: read next item (better: use a buffer manager to keep just the page you need open).
   - `close()`: close file.
2. *Projection*:
   - `open()`: open input.
   - `next()`: call `next()`, apply projection function.
   - `close()`: close input.
3. *Selection*:
   - `open()`: open input.
   - *Vertical fragmentation*: determined using `SELECT` clause (π), but must ensure `JOIN` and `WHERE` clauses can be processed.

12 Concurrency control

DBMS implements indivisible tasks (*transactions*):

1. *Atomicity*: all or nothing.
2. *Consistency*: consistent before $\rightarrow$ consistent after.
3. *Isolation*: indepenedent of any other transaction.
4. *Durability*: completed transactions are durable.

Transaction histories For a transaction T_n:

1. Begin transaction: b_n.
2. Read / write operations on objects: $r_n[o_j]$ and $w_n[o_j]$.
3. Commit / abort: c_n or a_n.

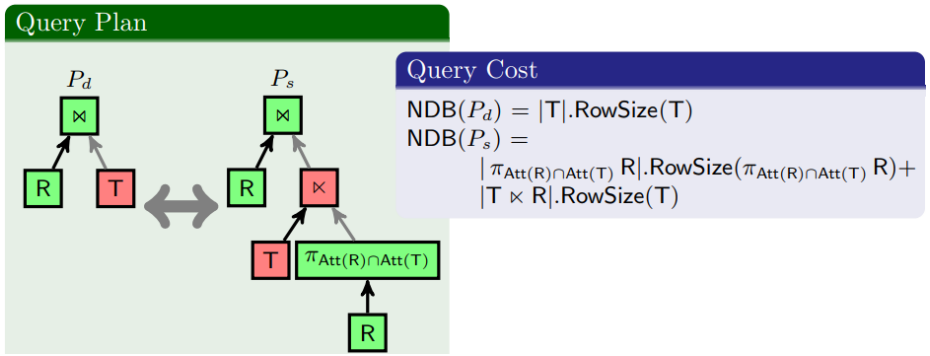
Cocurrency control Have to ensure:

1. *Serialisability*: a concurrent execution of transactions should have the same end result as some serial execution.
2. *Recoverability*: no transaction commits depending on data that has been produced by another transaction that has yet to commit.

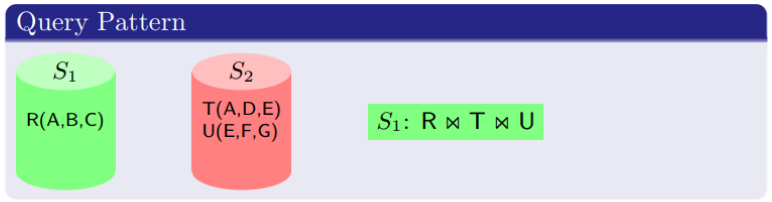
Anomolies

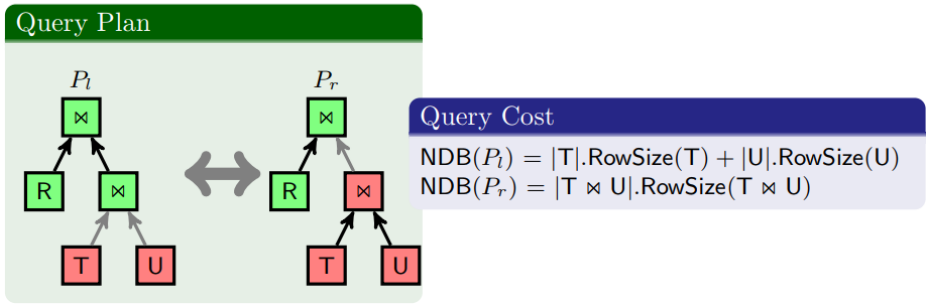
1. *Lost update*: $T_1$and T_2 update the same data element and one update is lost (overwritten).
2. *Inconsistent analysis*: T_1 acesses data before T_2 finishes working with that data.
3. *Dirty read*: T_1 reads a value that T_2 has written but not committed. Can cause non-recoverability.
4. *Dirty write*: T_1 writes to an object that T_2 has written to but not committed. Makes recoverability hard.

Direct join (high selectivity) ⇄ semi join (low selectivity):



Local join ⇄ remote join (denormalises data):





SQL queries over fragmented data

- *Horizontal fragmentation*: determined using `WHERE` clause (σ).

- `next()`: call `next()` on input until an item qualifies.
- `close()`: close input.

4. *Union*:
   - `open()`: open inputs.
   - `next()`: call `next()` on left input, or right input if we reach the end of LHS.
   - `close()`: close inputs.
5. *Difference*:
   - `open()`: buffer the right input, and open the left.
   - `next()`: call `next()` on left input until an item is not in the buffer.
   - `close()`: close left input.
6. *Cross* (pipeline-breaking):
   - `open()`: buffer the right input, and open the left.
   - `next()`: combine each left input with every right input (by keeping a pointer to buffer pos), then call `next()`.
   - `close()`: close left input.
7. *Cross* (streaming):
   - Return first results whilst buffering the RHS.
8. *Grouped aggregation*:
   - `open()`: read through the tuples, get the group keys, calculate the hash value, apply aggregate function and place in hash table.
   - `next()`: iterate through hash table slots.

Pipeline breakers Operators that produces the first correct output only after all input tuples have been processed:

- *Scan*, *selection*, *project*, *union* are not pipeline breakers.
- *Difference*, *grouping* are pipeline breakers.
- *Cross product*, *join* are often implemented as a pipeline breaker.

Buffer I/O in Volcano

1. *Scans*: sequenetial I/O of all pages in relation.
2. *Pipeline breakers*: `open()`: if buffer fits in memory, no I/O, otherwise sequential / random (assume one page per access) I/O.
3. *Pipeline breakers*: `next()`: if buffer fits in memory, no I/O, otherwise sequential I/O over buffer.

Function calls in Volcano

1. *Selections and prjections*: one to red input, one to apply predicate.
2. *Cross product*: one to read inner input, one to read outer input.
3. *Group by*: one to read input, one to extract group key, one to calculate each new aggregate value.

5 Joins

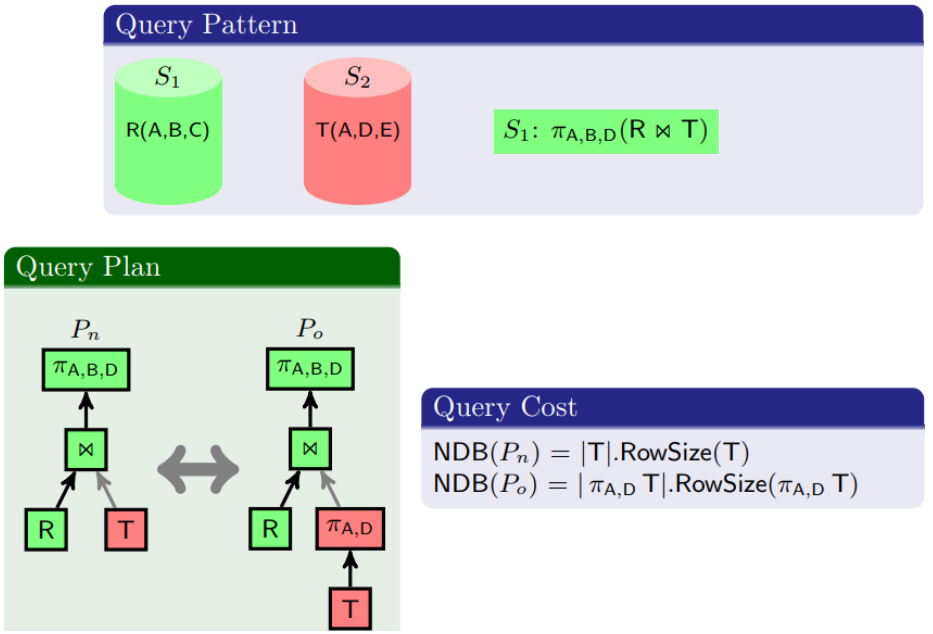
Nested loop join

- Simple.
- Sequential I/O.
- Trivial to parallelize (no dependent loop iterations).
- $\Theta(|left| \times |right|)$ with average effort of $\frac{|left| \times |right|}{2}$ assuming uniqueness

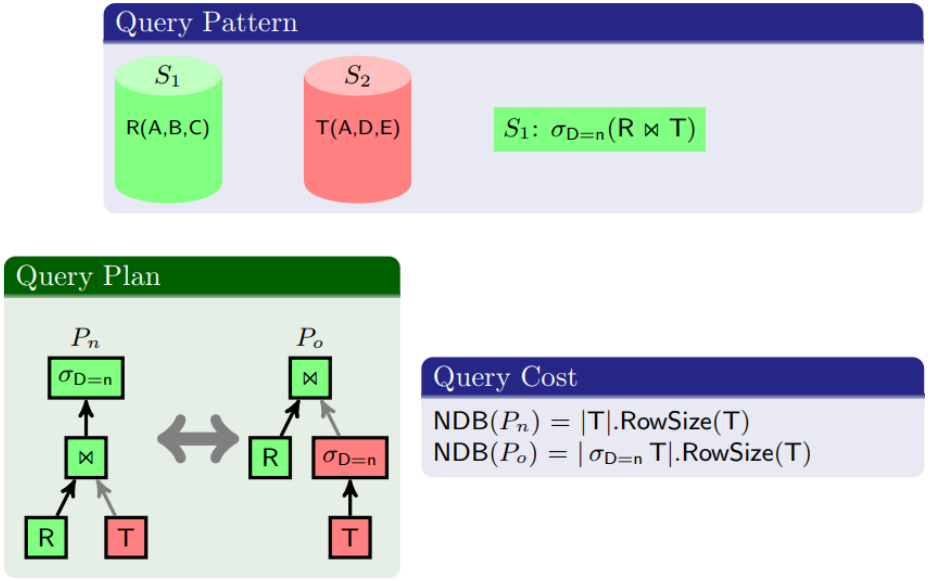
Sort-merge join Requires sortedness. Works for inequality joins.

- Sequential I/O in the merge phase.
- Tricky to parallelize.
- $O(|left| \times \log |left| + |right| \times \log |right| + |left| + |right|)$ assuming uniqueness.

Always push `PROJECT`s inside `JOIN`:



Always push `SELECT`s inside `JOIN`:

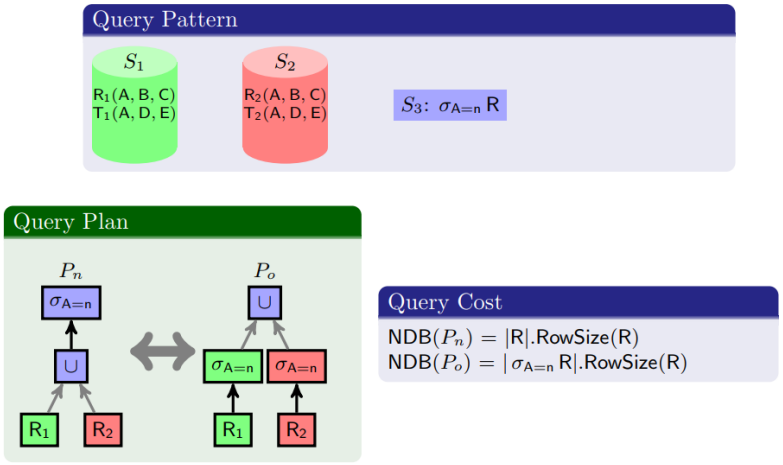


Distribution of transaction processing In a DDB:

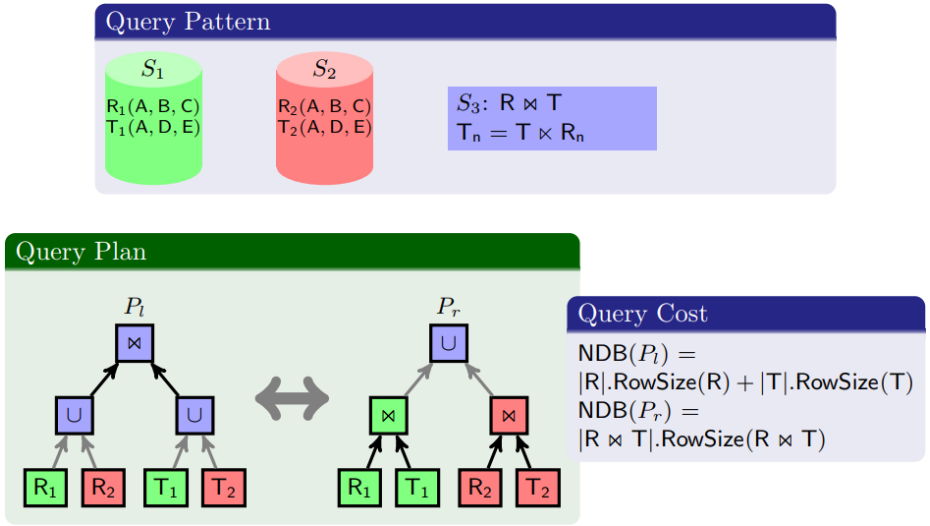
- Each site runs a *Local Transaction Manager*, scheduler and data manager.
- One or more sites run a *Global Transaction Manager*.
  - Transforms transaction into sub-transactions for each site.

RA Equivalences

`SELECT`s over horizontal fragmentation:



`JOIN`s over derived horizontal fragmentation:



Hash join Build-side is the buffered side in the hashtable, probe-side is the lookup the hashtable.

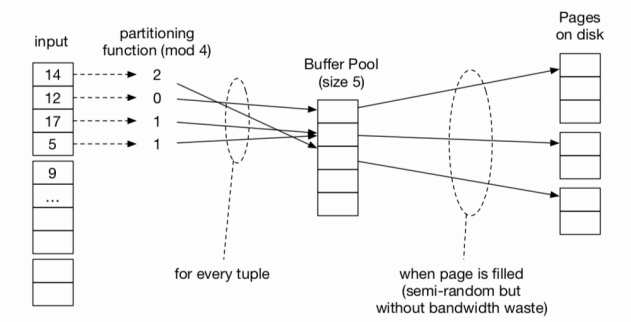
- *Hash function* (e.g. Modulo-Division) *requirements*:
  - *Pure*: no state.
  - *Known* ouptut domain.
  - *Contiguous* output domain: no holes in output domain.
  - Nice to property to have: *uniform*.
- *Handling conflicts requirements*:
  - Locality (to some extent).
  - No holes (probe all the output domain)
- Sequential I/O on the inputs.
- Parallelizable on probe side.
- $O(|build| \times |probe|)$ in the worst case, $\Theta(|build| + |probe|)$ in the best.
- *Disadvantages*:
  - Wastes space (over-allocated by at least 2).
  - People generally rehash (not hole-free), hashing costs lots of CPU cycles (can be more expensive than memory).
  - Probing is random access! If table doesn't fit into buffer pool $\implies$ lots of I/O.
- Often use buckets for slots.

Probing strategies Consider the distance from the original conflict

1. *Linear probing*: try increasing distance by 1 until no conflict.
   - Leads to long probe chains.
2. *Quadratic probing*: try doubling the distance until no conflict.
   - Good locality on first three probes and bad after.
   - Still likely to incur conflicts on first probes.
3. *Cyclic group probing*: Generate a sequence of numbers while making sure every number in range is generated.
   - $f(x) = (x \times g) \mod n$ where:
     - n is the size of the hashtable such that $n = p^k$ for odd prime p.
     - g is a primite root of p (difficult to find).

Partitioning Improves the performance of the probe-side of a join.

- Requires additional scans over the larger relation.
- Ensures sequential I/O.
- Easy to parallelize.
- We can partition the larger relation too, and only join overlapping partitions...



Block nested loops join Nested loops join, but scan per LHS page, rather than tuple.

Indexed nested loop join One side has an index, scan over the other side and use the index to get the matching tuple.

- Sequential I/O on the unindexed side.
- Quasi-random on the indexed side.
- Parallelizable over the values on the unindexed side
- $\Theta(|unidexed| \times |\log indexed|)$.

Which algorithm to use

- *Sort-merge join*: if relations are sorted or have similar sizes. Or if evaluating inequality-joins.
- *Index nested loops*: if one relation has an index.
- *Hash join*: if one relation is much smaller than the other (less than 10%).
- *Nested loops*: if one relation is tiny (< 20 values).
- *Block nested*: loops join for theta-joins.

Pig to MapReduce

- `FILTER` and `FOREACH` are implemented using Map.
- `GROUP` implemented by a Combine, then Reduce.
- All others implemented as just a Reduce.
  - If a Map process is after a Reduce, it can be put in the Reduce node.

Distributed hash join

1. Each Map node generates its bit of each hash bucket.
2. Reduce gets all files belonging to a particular hash bucket and performs join.

Replicated join

1. Replicate the entire RHS (smaller table) to all Map nodes holding the LHS.
2. Execute join as a Map process.

Skewed join

1. Uses a histogram of the frequency of join keys.
2. Histogram used to distribute the join over Reduce nodes. For keys with high frequency in LHS:
   (a) Distribute rows from LHS using round-robin.
   (b) Duplicate rows from RHS to multiple Reduce nodes.

Merge join Assumes both inputs are sorted.

1. Map nodes of LHS load required blocks from RHS.
2. Execute sort-merge join as a Map process.

11 Distributed query processing

DBMS componenets

1. *Transaction manager*: query processor plans and translates queries.
2. *Scheduler*: schedules primitive operators to obey ACID properties.
3. *Data manager*: interacts with memory maintains durability of transactions.

```
account_and_movement = JOIN account BY no LEFT,
                            movement BY no;
account_without_movement =
    FILTER account_and_movement
    BY movement::no IS NULL;
```

- *Group by*: produces a group column with the grouped-on value and a column containing a *bag* of tuples belonging to each group.

```
account_movements = GROUP movement BY no;
```

- *Flatten*: generate a row for each tuple in a bag:

```
movement_copy = FOREACH account_movements
                GENERATE FLATTEN(movement);
```

- *Aggregation*: `COUNT(a)`, `COUNT_STAR(a)`, `AVG(a)`, `MAX(a)`, `MIN(a)`, `SUM(a)`, or `DIFF(a, b)`:

```
movement_copy =
    FOREACH account_movements
    GENERATE group AS no,
             SUM(movement.amount) AS balance;
```

- *Cases*: Use nested statements:

```
account_credit_and_debit =
    FOREACH account_detail {
        credit = FILTER account_and_movement
                 BY amount > 0.0;
        debit = FILTER account_and_movement
                BY amount < 0.0;
        GENERATE group AS no,
                 COUNT(account_and_movement) AS ts,
                 SUM(credit.amount) AS credit,
                 SUM(debit.amount) AS debit;
    }
```

Pig optimisations

1. Project before a `GROUP BY`: less data to handle in the `GROUP BY`.

6 Bulk Processing

Turn *control dependencies* into *data dependencies*. No function calls (no jumps $\Rightarrow$ CPU efficient).

Page access probability from selectivity: $1 - (1 - s)^n$

- *Materialise all*: write down entire tuple in output array on each operator.
- *By reference*: return array of reference to the Tuples in the table.
- *By reference using column storage (DSM)*: same idea but optimises predicate evaluation by having more values fit on a page

7 Secondary Storage

Indices

- *Clustered or Primary Index*: store the tuples of a table, at most 1,
- *Unclustered or Secondary Index*: store pointers to the tuples of a table.

Hash indexing Hash table rom key $\rightarrow$ position.

- *Maintenance*:
  - Overallocate by a lot.
  - Need to rebuild if fill factor is too high (expensive even with consistent hashing).
  - On delete need to put a marker in the hashtable for future probing.
  - Load spikes ☹.
- *Useful for*:
  - Equi-joins, aggregation on specific keys.
  - Filter on specific keys.

Bitmap indexing A bitvector for each distinct value in a column (few distinct values).

- *Run-length encoding*: replace consecutive occurrences with length of the chain.
  - Works well on high locality data.
  - Requires sequential scan to find values at a specific position.
- *Useful for*:
  - Can reduce bandwidth needed for scanning a column.
  - Can use arbitrary conditions (make disjoint sets of values called binned bitmaps).

Foreign-Key indices Specifies that there is exactly one value in the PK column of the other table

- Need to maintain integrity, DBMS on insert/update need to make sure the value doesn't already exist.
- Implemented using a pointer because it means it joins to exactly one row.
- *Advantages*:
  - Low space requirement.
  - Instant joins.
  - Insignificant effort added on insert/update.

B-Trees Reduce load spikes by using a tree.

- *Advantages*: Support ranges, self balancing
- *Disadvantages*: Complex, leaf pointers aren't used, most of the data lives in the leaf nodes but not all of it (need to go up).

Insertion:

1. Find the right leaf-node to insert.
2. If the node overflows, split the node into two halves, choose the median to go in the parent.

B+-Trees Keep all key and pointers to data in the leaf nodes, replicate the keys above, link the leaves like a linked list.

- *Advantages*: scans over ranges are trivial.

MapReduce

1. *Load*: items from data nodes to Map nodes.
2. *Map*: nodes perform map function on each item.
3. *Combine*: partially calculate the Reduce on the Map nodes.
4. *Shuffle*: send each item to the correct Reduce node.
5. *Reduce*: nodes perform reduction function on incoming items.

Pig Latin

- *Load*: make data source available as relation:

```
account = LOAD 'file.tsv'
          AS (no:int, type:chararray, rate:float);
```

- *Strore*: execute script and store result:

```
STORE account INTO 'copy' USING PigStorage(',');
```

- *Project*:

```
rate = FOREACH account GENERATE rate;
distinct_rate = DISTINCT rate;
```

- *Filter*:

```
account_with_rate = FILTER account BY rate > 0.0;
```

- *Cross*:

```
branch_with_account = CROSS branch, account;
```

- *Equi-join*:

```
branch_and_account = JOIN branch BY sortcode
                          account BY sortcode;
```

- *Union*:

```
id = UNION branch_sortcode, account_no;
distinct_id = DISTINCT id;
```

- *Difference*: use a left join:

Replication Copy data objects between sites.

- Queries may run on any site.
- Updates must be written to all sites.
- Usually more reliable (individual copies may go down).
- Usually faster (by reducing load on each site).

Migration Move data to where its used.

10 Big data

Data models

- *Key-value.*
  - Schema-less.
  - Very limited querying capabalities.
  - Useful for caching.
- *Document store*: document (semi-structured) data model (e.g. JSON).
  - Schema-less.
  - Support queries searching field values.
  - Use MapReduce for OLAP.
- *Wide column*: table data model with easy addition of new columns. Columns may be put in families.
  - Schema-less.
  - Support queries searching field values.
  - Use MapReduce for OLAP.
- *Relational*: relational data model.
  - Schema based.
  - Support queries searching fields and performing joins.
  - ACID properties of transactions.
- *Graph*: nodes and edges (e.g. RDF).
  - Schema-less.
  - Limited querying possible.

Materialized views

- Basically alias to a query.
  - Some systems actually stores the result of the query and modify it on modifications of the underlying relations (expensive).
  - *Useful for*: running the same query returns result instantaneously.

8 Query Planning and Optimisation

Start with a correct plan and apply equivalence rules.

- From the root of the plan, apply pattern to start new plan and start process again until you traverse without applying any transformation.
- *Logical*: algorithm-agnostic.
- *Physical*: algorithm-aware.
- *Rule-based*: data-agnostic.
- *Cost-based*: data-aware.

Logical rule-based optimization Simple and portable, but sometimes wrong, infinite cycles, missing rules.

- Selection and projection **pushed** through joins if only applied to one side
- **Operator reordering** (joins, selection, unions) using a heuristic like $s(==) < s(<) < s(<>)$.

Cost-based optimisation Cost metric could be number of tuples produced.

- *Estimations*: like selecting one value: $\frac{1}{\text{disctinct values in column}}$ and recording distinct values are brittle.
- *Statistics*: use histograms $\frac{\text{ocurrences of a value}}{\text{total tuple count}}$ and evaluate query on histogram first.
- *Multidimensional histograms* used to resolve attribute correlation.

Physical optimisation

- **Example metrics:** number of function calls, sum of all produced tuples (incl. materializations), number of page faults (I/O), CPU costs, max(I/O, CPU), total intermediate size, ...
- **Counting cost is hard:** different algorithms have different costs on all the above, how do you weight them?

Rule based physical optimisation Focused on hardware (parallelism, cache-conscious partitioning).

Cost-based physical optimisation Generally access path selection: from base table, indices, etc.

9 Distributed databases

CAP theory No distributed system can maintain all three:

1. *Consistency*: all nodes see the same version of data.
2. *Availability*: system always responds within fixed upper limit of time.
3. *Partition tolerance*: system always gives correct response even when messages are lost.

Approaches

1. *Heterogeneous DDB*: varied database techology, managed by different DBAs, designed at different times.
   - Requires a *commmon data model*
   - Need to perform *schema integration*.

   (a) Transform *local schemas* to a standard data modelling language.
   (b) Filter to get *export schemas*.
   (c) Construct *global schemas*.
   (d) Filter to get *external schema* for applications.

   - *Data warehouse*: materialise copies of data with global schema.
   - *Mediator architecture*: gives live results, but higher latency and demand on systems.
2. *Homogeneous DDB*: same technology, one DBA, designed at same time.
   - Different sites each have *local schema*.
   - Combine to form a single *global schema*.
   - Different *external schemas* for different applications.

Fragmentation / sharding Split data objects between sites, queries and updates correspondingly distributed.

- Usually less reliable (each database can go down).
- Usually faster (parallelism in horizontal fragmentation).

1. *Horizontal fragmentation*:
   $$R = R_1 \cup \cdots \cup R_n$$
   - Split rows using σ:
     $$R_1 = \sigma_{P_1} R, \ldots, R_n = \sigma_{P_n} R$$
   - Derived horizontal fragmentation splits rows using $\ltimes$:
     $$R_1 = R \ltimes S_1, \ldots, R \ltimes S_n$$
   - Fragmentation rules:
     (a) Must cover all possible values.
     (b) Should not involve things that change!
     (c) Should be easy to fragment on.
2. *Vertical fragmentation*: A loss-less join decomposition:
   $$R = R_1 \bowtie \ldots \bowtie R_n$$
   - Splits rows using π:
     $$R_1 = \pi_{\text{attrs}_1} R, \ldots, R_n = \pi_{\text{attrs}_n} R$$
   - Fragmentation rule should include a key.