1 Join Algorithms

For all of the algorithm discussions here, consider two relations R and S, where $|R| \leq |S|$. $|M|$ is the size of primary memory.

1.1 Sort-Merge

1. Generate sorted runs of the tuples in R and S. These runs are (on average) twice as long as can fit in memory at once ($2 * |M|$).
2. Sort these runs using n-way merges, with n as large as possible (to decrease the number of required merging phases and IO time). Ideally, only two phases are required; this is possible when $|M| \geq \sqrt{|S|}$. One buffer page is required for each run.
3. Generate output tuples as the R and S tuples fall into sorted order.

1.2 (Classic) Hash Join

When calculating the sizes of any table, a fudge factor F is used to represent hash table space overhead.

1. Create a hash table based on the smaller relation R, hashed by the join attribute.
2. For each tuple in S, hash the join attribute and look up the result in the hash table. Create an output tuple if a match is found.

Of course, this works well if the hash table for R can fit into primary memory. When this is not the case:

1. Use a (different) hash function to partition both R and S into disjoint sets $R_1, R_2, \cdots, R_n$ and $S_1, S_2, \cdots, S_n$. Note, this is an equal number of sets that are not equally sized (it should be expected that the average R_i is smaller than the average S_i, since $|R| \leq |S|$.
2. Perform the hash join for each of these disjoint sets $R_i \bowtie S_i$.
3. Concatenate the resulting sub-tables.

1.3 Simple Hash Join

Identical to Classic Hash Join if R fits into memory.

Otherwise, essentially perform Classic Hash Join by dealing with each partition H_i of R and S (as determined by the hash function) in turn, with bucket overflow going to disk.

1. Create a hash table of the tuples of R that fall into H_i, writing overflow onto disk.
2. Probe table and generate output tuples.
3. Increment i and repeat.

The main disadvantage is that R and S are both scanned many times.

1.4 GRACE Join

This is a two-phase algorithm, which starts off by partitioning R into approximately equal sized buckets that each fit into memory.

1.4.1 Phase 1

1. Choose a hash function h that will partition R as described above.
2. Scan R and use h to place tuples into output buffers.
3. Scan S and use the same h to place its tuples into output buffers.

1.4.2 Phase 2

This section is repeated for each i from zero to the number of partitions of R. A different hash function is used for this section, h'.

1. Build a hash table in memory for R_i. (Overflow discussed later)
2. Using h', probe the hash table for each tuple of S_i, writing output tuples as necessary.

1.4.3 Dealing with Overflow

There are several available strategies:

- Re-partition with yet another hash function during phase 2, as needed.
- Over-partition in phase 1, merging these small partitions on the fly as space permits.
- Use statistics collected on the data to estimate a good hash function up front.

1.4.4 Notes

Always scans R and S twice (but never more than twice as in Simple). This is a loss if most of R can fit in memory.

1.5 Hybrid Hash Join

This is very similar to GRACE, except the first hash table (for R_0) is kept in memory during phase 1 and probed while S is being partitioned. This way, R_0 and S_0 never need to be written to disk.

1.6 Algorithm Comparisons (Sort Merge versus Hash Join)

Hash join is typically a very good choice if one of the relations being joined fits entirely in memory.
Otherwise, Sort merge basically always wins if:

- The relations are already sorted.
- There are many duplicates or the data is catastrophically non-uniform. Sort merge can also just be used on a subset of bad partitions, if necessary, and if the rest would benefit from hash join.
- Output needs to be in sorted order.

1.7 Other Optimizations

- **Bloom Filters**
  Build a Bloom Filter based on the join attributes in R, and use that to augment the hash table. The filter allows you to pre-reject tuples in S, since it will trivially tell if there is definitely no match in R.

- **Semi-Join ($\ltimes$)**
  Let the $\pi(R)$ be the projection of the joining attribute of R, and let the semi-join be denoted $S \ltimes R = S \bowtie \pi(R)$. The semi-join optimization is, then:

$$(S \ltimes R) \bowtie R$$

  This is equivalent to $R \bowtie S$, but can save significant IO time if R has many attributes that are not being joined on, but are required for the final result.

  This procedure essentially uses the semi-join to filter S without carrying around tuple baggage from R.

2 System R Optimizations

These optimizations largely center around query planning and join ordering to minimize IO costs. This is quantified in a "Selectivity Factor" assigned to each operation in the query. The process also takes into account "interesting orders", where the result of some operation might yield an order that would be useful later (either for another join or as in a sorted order that was requested in the query for the final result set).

Some notation for this section:

- $|R|$ is the number of pages in R.
- $||R||$ is the cardinality of R.

2.1 Optimizer Inputs

- Search Space
- Cost Estimation (statistics, histograms - equiwidth, equidepth)
- Enumeration Algorithm

2.1.1 Statistics

Statistics Tracked:

- Cardinality of relation R.
- Number of pages holding tuples of R.
- Fraction of pages in the segment holding tuples of R.
- Number of distinct keys in index I of R.
- Number of pages in index I of R.

These statistics are updated periodically, and are not always accurate. Their primary use is to assign selectivity factors to operations. When no statistics are available, arbitrary selectivity factors are assigned (with the intuition being that only a small relation would have no statistics).

One compact representation of key distributions over tables is the *histogram*. These can be either equi-width (each bucket covers the same amount of key space) or equi-depth (where each bucket contains the same number of keys, but covers a variable amount of key space). Equi-depth histograms are more useful in dealing with significantly skewed data. These do not provide any sort of joint distribution information by themselves.

2.2 Interesting Orders

An order is interesting if it is induced by the query itself (by ORDER BY or GROUP BY clauses), or if it would beneficially feed into a later stage (such as providing tuples in a sorted order for a sort merge join).

Cost calculations must take into account the final interesting order (or any intermediate sorting costs) for branches of the optimization tree that do not have a cheap interesting order available.

2.3 Join Optimization Algorithm

The System R optimizer simply uses bottom-up dynamic programming to construct a search tree through the optimization space. The final query plan chosen is the cheapest (calculated by traversing the tree from the leaves to the root).

First (step zero), find the cheapest access path for each relation on any given interesting order. Keep the cheapest for each interesting order, along with the cheapest uninteresting order (if it is cheaper than the interesting orders).

1. For each join $\bowtie_i$, compute the cost for choosing each access path.
2. Retain the cheapest plans for each interesting order, along with the cheapest plan that does not produce an interesting order (unless the cheapest plan that results in an uninteresting order is more expensive than an interesting order). If a particular interesting order is required for the result of $\bowtie_i$, factor in the cost to sort for plans that do not provide this interesting order.
3. Create a new set of relations to be joined, $\bowtie_i'$.
4. Go to 1 unless there are no joins left.

The algorithm first finds all best ways to join all pairs of relations, $\{\bowtie\}_2$. It then finds all best ways to join all sets of three relations (by extending the joins of size two with one more joined relation), and so on until all relations have been joined. When the iteration is over, a tree of query plans is in place and the cheapest can be selected.

At each node in the tree, estimated result set cardinalities are maintained. These are only estimates, and errors thus propagate up the tree. These estimates feed into the next round of cost estimations. Errors can be mitigated by re-optimizing the search tree on-the-fly as the query progresses.

2.3.1 Join Order Heuristic

System R uses this heuristic to trim the search space when building the optimization tree: only consider join orders which have join predicates relating the inner relation to the other relations already participating in the join (where possible).

This essentially means that Cartesian products are pushed to the end of the computation.

2.3.2 Optimization Paths not Considered

- **Bushy Joins**
  These require fully materializing at least one intermediate result (which means IO), so they are not considered by System R.

  An example of when a bushy join could be a major win would be in a star topology with four small satellite tables and a huge central table where there are Cartesian joins between the small tables and equijoins against the huge table. Computing the full cross product of all of the small tables and only traversing the huge table once could be a massive IO win.

- **Cartesian Joins**
  System R does not consider these because they are typically undesirable, and would expand the search space considerably.

A full Cartesian join is typically bad, but can be a win in certain query patterns. One example is a star join, where a huge central relation is joined to several (highly selective) smaller relations. Joining each small relation to the large relation in series would require several scans of the huge relation. Performing the Cartesian join among the smaller relations, however, produces one (somewhat larger than the original small relations) relation that can be joined against the huge relation in one pass, saving lots of IO overhead.

2.4 Nested Queries

Sub-queries that do not capture any context of their enclosing query need only be evaluated once (since their environment does not change). Both single values and sub-lists can be optimized in this manner.

Queries that *do* capture some aspect of their enclosing environment are referred to as *correlated queries*. In theory, correlated queries must be re-evaluated for every time they are encountered. Some of these re-evaluations can be avoided by sorting on the captured field, and saving the result of the evaluation until the field changes (and a re-evaluation is unavoidable).

2.4.1 Preserving Duplicates and NULLs

If a sub-query has an aggregation, duplicates and NULLs in the outer query can be preserved by using a *left outer join*.

2.5 Other Optimizations

These are a few other optimizations covered in Chaudhuri98.

2.5.1 Join/Group By Interaction

It can be profitable to push Group By operations earlier, especially if they eliminate a large number of records (which no longer need to be joined, possibly making one relation fit into memory). Group By can also be very cheap if an appropriate index is available.

2.6 Optimization Architectures

Besides the simple bottom-up tree building approach of System R, several other optimization strategies are also common.

- Algebraic query rewriting. These are based entirely on rules engines that apply algebraic transformations when they are profitable.

This style allows rules to be added to the engine without a complete rewrite, and can optimize things besides joins.

3 Buffer Management

Database people consider traditional memory management techniques unsuitable for database workloads, so they cooked up several more context aware alternatives for their memory managers.

In all cases, a Load Controller process allows queries into the system conditionally, depending on available resources (queries that will take more resources than are available will wait until resources free up). This does not make any inherent starvation guarantees. If the Load Controller makes a bad estimate and a query starts to take too many resources, the controller can suspend that query.

3.1 Domain Separation Algorithms

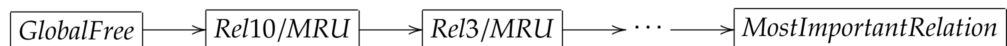
This was an earlier approach that used separate buffer pools for related workloads. As a motivating example, the internal nodes of a B-tree are much more likely to be accessed than any particular leaf or data node, so leave one pool for internal nodes, with leaves and data in another.

This does work to an extent, but it provides only a static view of the importance of any given page; this is a problem since some pages become more or less important in various contexts. Additionally, it allows queries to interfere with one another (say, index pages from one query causing index pages from another query to be evicted).

3.2 "New" Algorithm

This algorithm seemingly introduced per-relation buffer pools. The intuition involved was that the importance of a given page is not an inherent property, but more of a property of the relation to which it belongs.

The idea is to maintain a linked (priority) list of buffer pools, one per relation plus a global free pool at the head of the list. When a page is needed (in response to a page fault), the list is searched from the top (global) to the bottom until a free page is found; the page is placed in the pool of the relation that requested it.



Obviously, the order chosen for relations (itself chosen statically) is of critical importance, with less important relations (closer to the global pool) losing pages first when memory is low. The list is only traversed if all pages in the currently examined relation are in use, so relations at the head of the list get their pages kicked out frequently.

Additionally, performing the required linear search can be very expensive.

3.3 Hot Set Algorithm

In this model, each query gets its own buffer pool, sized according to the Hot Set model. Each of these buffer pools is managed according to LRU. Queries are allowed into the system if their hot sets do not exceed the remaining buffer capacity.

Hot sets are identified by looping behavior in queries. This model tends to over-allocate memory (to ensure that there is no contention among queries for data structures).

3.4 DBMIN

This algorithm is based on the Query Locality Set Model (QSLM). This model decomposes database operations into a small set of categories that each have their own caching and allocation behavior. These categories are used to allocate buffers on a per-file instance basis; essentially, each query gets its own set of per-file buffers (but these can be shared for efficiency).

3.4.1 QLSM

Name	Buffers Required	Policy
Straight Sequential	1	-
Clustered Sequential	Max Cluster Size (estimate based on statistics)	LRU *or* MRU
Looping Sequential	All	MRU
Independent Random	1 or b (from Yao's Formula)	-
Clustered Random	Same as Clustered Sequential	-
Straight Hierarchical	1	-

4 Locking

The primary contribution of the locking paper was finer-grained locks, especially regarding the intent locks. The existing locks are Shared (Read) and Exclusive (Write). New locks are Intent to Share (IS), Intent to Write (IX), and Shared with Intent to Lock Exclusively (SIX).

4.1 Access Modes

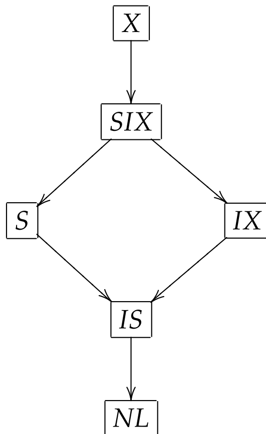
Acquiring an S or X lock on a node implicitly locks all of the children of that node. IX, IS, and SIX locks do not lock anything on their own, only declare intent to do so later; they basically operate by denying other forms of intent lock (or stronger incompatible locks).

- Descendants of nodes locked in IS mode can only be locked in either IS or S modes.
- Descendants of nodes locked in IX mode can be locked in any mode.
- Descendants of nodes locked in SIX mode can be locked in IX, X, or SIX modes.

4.1.1 Compatibility

	NL	IS	IX	S	SIX	X
NL	Yes	Yes	Yes	Yes	Yes	Yes
IS	Yes	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	Yes	No	No	No
S	Yes	Yes	No	Yes	No	No
SIX	Yes	Yes	No	No	No	No
X	Yes	No	No	No	No	No

The partial ordering among locks:



The S and IX locks are not really comparable.

4.2 Degrees of Consistency

	Locks	Recoverable	Dirty Reads	Repeatable Reads	SQL Isolation
Degree 3	Long X, Long S	Yes	No	Yes	Serializable or Repeatable Reads
Degree 2	Long X, Short S	Yes	No	No	Read Committed
Degree 1	Long X	Yes	Yes	No	Read Uncommitted
Degree 0	Short X	No	Yes	No	None

Table Notes:

- Dirty Reads: $W_2 \rightarrow R_1$

- Repeatable Reads: $R_1 \rightarrow W_2 \rightarrow R_1$
- Serializable guarantees no phantoms
- Degree 0 isn't really much of a transaction

4.3 Physical Granularity

Instead of trying to guess at query planning time at what level to lock (tuples, pages, files, tables, etc), the lock manager can simply notice when a transaction is locking too many rows and *upgrade* locks.

4.4 The Locking Protocol

Reads must lock at least one path from the "root" resource (the database). This path is along a DAG.

Writes must lock *all* paths from the root to the records they are modifying.

Locks are acquired root to leaf, and released leaf to root.

- **To Acquire S or IS Locks:**
  All ancestors must be locked IX or IS.
- **To Acquire X, SIX, or IX Locks:**
  All ancestors must be locked IX or SIX.

4.5 Phantoms: Serializable versus Repeatable Reads

Phantom records are a product of locking resources at the *physical* layer, rather than the *logical*. A simple example of the problem follows:

1. T_1 locks the *pages* required for its query in S mode.
2. T_2 adds some tuples that would affect T_1, but they are added into a new page (of which T_1 was not aware, since it had not yet been allocated).
3. T_1 does not see these new records, and it does not have the consistency guarantee that it would like.

Resolving this type of conflict requires higher level locks, either on a range in an index, if one exists, or through predicate locks. Predicate locks would make any transaction attempting to acquire a lock that violates the predicate wait.

4.6 B-Tree Locking

While the lock ordering on database resources must be well-ordered and strictly two-phase, locking on B-Trees is somewhat more relaxed.

One major reason for this is that B-Trees are not directly exposed to the user, and offer equivalent functionality, even in different structural configurations (assuming B-Tree invariants are maintained with proper latching).

One B-Tree invariant that is often ignored for performance reasons is the load factor; when a node drops below 50 percent utilization, no coalescing is performed. In most database workloads, the database continually grows (with local shrinkage that is later filled in).

4.7 Lock Coupling

Lock coupling minimizes the number of locks held at any given time, and is enabled by the observation that, if a node is *safe*, none of its parents will need to be modified for the current operation. A node is *safe* if there is room to insert a new node without causing a split; this is very frequently the case.

4.7.1 Read Lock Coupling

As usual, the case where a transaction is only reading data is simpler. In this case, the process can "spider" down the tree, acquiring a lock on a node, descending to the relevant child, acquiring that lock, and releasing the parent lock.

This trivially frees higher nodes for other processes.

4.7.2 Write Lock Coupling

Writing processes need to pay attention to node safety. If a node is safe, the operation can give up all locks on parents of that node.

A more optimistic variant of this scheme is to descend all the way to the leaf node that needs to be modified, and acquire locks *upwards* until a safe node is reached.

4.8 (R)ead with (I)ntention to (W)write Locks

These locks are like SIX locks in the granularity paper; acquiring this type of lock on a node announces that you will be reading this node, and possibly writing some children of this node. This allows other read locks to be placed on a node without interference. RIW locks are not, however, compatible with each other.

When writing is required, the RIW lock can be "upgraded" (closer to the write location) into a real write lock, and behaves as normal thereafter.

4.9 Sibling Link Trees

The optimization shown in the B-Tree locking paper (Lehman and Yao) allows B-Tree operations to require at *most* three locks (and two for the vast majority of the time). Their main insight is that the actual shape of the B-Tree and location of data is irrelevant, as long as the lookup procedure can, at any time, find what it is looking for (if it is in the tree).

There is only one underlying problem: when a piece of data is being inserted and a node needs to be split, a search must block until the reorganization is finished (this involves copying half of the data out of one of the pages, and bookkeeping) or it will (potentially) miss an element that was in the process of being moved.

The solution is two-fold, and requires minimal additions to the tree: use a High Key and Sibling Links.

4.9.1 High Key

The high key is a bookkeeping key added to the end of every internal B-Tree node; it is basically a cache of the *highest* key in the right-most sub-tree of the current node.

The use of this key is outlined later.

4.9.2 Sibling Links

These links are between B-Tree nodes on the same level; their primary use is to connect the two parts of a node that is in the process of being split. When a node is split, the parent of the left node is correct, but the right node is not yet known to its parent, and a typical search would miss it. With a slight modification to the search protocol, this sibling link allows the search to find these "orphan" nodes before they are fully consistent.

4.9.3 Usage

The search algorithm is modified as follows: if the target item is not found on a page, try to follow the sibling pointer **if and only if** the high key is *greater* than the key we are searching for. This means that the node was split and what we want is probably on the next page (or somewhere along the page chain).

This allows the parent node to be updated (relatively) lazily. Lock coupling (or one of its variants) can be used to walk these potentially long sibling chains.

The worst case locking scenario is that three nodes are locked: the data node being modified, its parent (which is in the process of being split), and one more node along the chain. The second lock can be dropped relatively quickly (after the lock down the chain in the meta-node is acquired).

5 Oracle Concurrency Control

Timestamp based (called the System Change Number - essentially a Lamport clock). Not quite as strong as Repeatable Reads, but stronger than Read Committed.

Oracle uses no read locks, so Readers can never block Writers.

The SCNs impose an ordering on transactions, and transaction integrity is preserved by essentially snapshotting any data before it is changed, working only on a local copy.

5.1 Locking

All Oracle locks are at the tuple level; as locks reside directly on data pages, the overhead is not large.

5.2 Rollback Segments

Older versions of data are stored rollback segments, which are tagged with the SCN that created them. If the live version of a piece of data has an SCN greater than that of a transaction that needs it, the reading transaction instead consults the relevant rollback segment to find the older version of the data.

Readers, then, always read the rollback segment (or live data) with the largest SCN that is less than their own.

5.3 Simple Writes

- Start
- Get SCN (atomically)
- X-lock required rows
  - Write a rollback segment
  - Update locked rows
  - Flush redo logs
- Committed

The sub-list can be viewed as the sequence required to write a single row.

6 Optimistic Concurrency Control

Main idea: in workloads where conflicts are rare, locking is a huge overhead that is nearly never necessary. Instead, attempt to act (in an isolated way), and check for conflicts after the fact (but before committing).

These methods are deadlock free (since there are no locks), but care must be taken to prevent starvation.

6.1 Transaction Phases

All transactions are divided into three stages:

- Read
- Validate

- Write (not mandatory)

The Read phase does not necessarily only involve reading, but any writes performed during the Read phase are performed on local copies of data. Validation ensures that any changes during the Read phase will not violate the integrity of the database (the transaction will be restarted if Validation fails).

The Read and Write phases are straightforward. During the Read phase, data can be read as needed; if any object needs to be written to, a local copy is made which can be modified (and is not visible to any other transaction). During the Write phase, these local copies are made globally accessible (under a lock). The Read phase constructs both Read and Write sets, which are used later in the validation phase.

6.2 Validation

All of the complexity of Optimistic CC is in the Validation phase; the proposed methods use serial equivalence as the criteria to determine transaction validity[1].

In order to determine if such a serial equivalence exists, each transaction is assigned a *transaction id* at the end of its Read phase. Any one of the following conditions is sufficient to prove that a pair of transactions T_i and T_j are serializable:

1. T_i completes its Write phase before T_j starts its Read phase (the trivial condition).
2. $\mathit{WriteSet}(T_i) \cap \mathit{ReadSet}(T_j) = \emptyset$ and T_i completes its Write phase before T_j begins its Write phase.
3. $\mathit{WriteSet}(T_i) \cap \mathit{ReadSet}(T_j) = \emptyset$, $\mathit{WriteSet}(T_i) \cap \mathit{WriteSet}(T_j) = \emptyset$, and T_i completes its Read phase before T_j.

When transaction ids are assigned at the end of the read phase, $t_i < t_j$ (where t_i is the transaction id of T_i) is automatically satisfied. Additionally, if transaction ids were assigned at the beginning of the read phase, fast transactions that start after slow transactions would need to wait for the slow transactions to finish before they could be validated, killing concurrency.

6.2.1 Read/Write Sets

The read and write sets generated during the Read phase of each transaction need to be maintained as long as there exists some transaction that *began* before the sets were created. This means that the read and write sets of fast transactions, which started *after* some long-running transaction T, are maintained until T is done with them in its Validation phase.

The space for storing this history, however, is finite. A long transaction could cause this space to fill up, and it would no longer be possible to validate it (or process more transactions). In this case, the long running transaction must be killed (since it cannot be validated). It can try again as a new transaction; if it persistently fails, though, it would face starvation.

If a long running transaction is starving, the simple solution is to simply allow it to run to completion under a critical section if some failure threshold is passed. This could re-use the lock around the validation phase.

6.2.2 Serial Validation

The serial validation scheme is simple and validates only using conditions (1) and (2), discussed above.

Acquire Lock

- Get Transaction ID (and increment the global counter source)
- Validate either (1) or (2)
- If Valid, Write

[1]This is a fairly simple notion with a fancy name. In order for a set of transactions $T_1, T_2, \cdots, T_n$ to have serial equivalence, there needs only be some serial ordering of transactions such that $State_{final} = T_n \circ \cdots \circ T_2 \circ T_1 = State_{initial}$

Release Lock

Note, the entire validation (**and** the write) is performed in a critical section.

As an optimization, the transaction ID can simply be read (and not claimed) before the critical section. Then, transaction IDs up to that ID can be validated *outside* of the critical section. If the validation fails, the transaction in the validation phase can be restarted without ever having to enter the critical section. This process can be repeated as many times as desired before entering the critical section and performing the final comparisons (against transactions that completed since the last round of pre-validation).

Note that if a transaction only reads data, it does not need to enter the critical section at all after it performs one pre-validation.

6.2.3 Parallel Validation

The parallel validation validates against all three possible criteria, and works on a finer grain. The method here also uses the optimization discussed above.

Acquire Lock

- Read Transaction ID (do not increment)
- Add self to the Active Transaction Set

Release Lock

- Check (1) and (2) for each transaction from the Start Transaction ID (recorded before the Read phase) to the End Transaction ID (read as the first step of validation)
- Check (3) for each transaction in the *Active* Set
- If Valid, Write

Acquire Lock

- Increment Transaction ID source
- Remove self from the Active Set.

Release Lock

The checks against conditions (1) and (2) are identical to the serial validation method, and can be repeated up to new tentative transaction IDs outside of the critical section.

Failures can cascade from transactions in the active set that invalidate one transaction, but are themselves invalidated later (meaning the original invalidated transaction would have been fine).