1 Transactions

1.1 ARIES

Of the ACID properties, ARIES addresses atomicity and durability (in the face of system failures). The recovery method is built on logging (in particular, write-ahead-logging). To provide a relatively orthogonal system, ARIES imposes only one important requirement on the format of redo operations: they must be *idempotent.*

The idempotence requirement makes logical logging somewhat difficult, since most forms of logical logging are inherently not idempotent (consider "add 5 to attribute A"). In exchange, ARIES gives *repeatable history* guarantees, which it uses to provide durability.

1.1.1 (Write-Ahead) Logging

The primary mechanism that ARIES uses to maintain data integrity is write-ahead logging. Logs are flushed on commit, and *must* be written before the associated data is updated (hence "ahead").

Each log record conceptually has a Log Sequence Number (LSN), a log type, and, for undo logs, some description of the change being made. The offset of the log record into the log can be substituted for an explicit LSN, since it is automatically a monotonically increasing sequence. Typical space saving measures: delta compression and logical logging.

The log itself is essentially a series of linked lists (one list for each transaction) interleaved. Links are provided backwards in time through the log by a PreviousLSN field in each log record, which is used when replaying the log. Further, compensating log records (which record an *undo* operation) have an additional pointer to the operation that they undo.

1.1.2 Page Management

There are two tools for managing pages of memory that can either be useful or harmful:

Forcing: The force operation immediately commits all dirty pages to disk. Doing this often leads to poor response times.

Stealing: Stealing an active page writes its contents to disk so that the page may be re-used for something else. This is something like aggressive swapping. Not allowing this reduces throughput.

Ideally, a system wants stealable pages without frequent forcing. Each page has at least one piece of metadata: the LSN of the last modification to the page. This is used during recovery to determine if a given modification could have possibly modified the page at all.

1.1.3 Checkpoints

Checkpoints reduce the amount of log replay required when restarting, since only log segments up the last checkpoint need to be replayed; checkpoints take a long time to complete, though, and so are unsuitable for online systems. ARIES instead uses *fuzzy* checkpoints, which give a view of the data consistent with when the checkpoint was started. Two tables are used to facilitate fuzzy checkpoints:

- Transaction Table (*xid*, *LastLSN*)
- Dirty Page Table (*xid*, *RecLSN*)

The RecLSN is the LSN of the *first* log record that made a change to a page while the fuzzy checkpoint was in progress. The checkpoint procedure is as follows:

1. Insert a `BEGIN CHECKPOINT` record into the log.
2. Inside of a critical section, take a snapshot of the Transaction and Dirty Page tables.
3. Insert an `END CHECKPOINT` record into the log. This record contains the snapshots of the Transaction and Dirty Page tables.

4. Record the LSN of the `BEGIN CHECKPOINT` record as the latest *master record.*

The recovery procedure can skip right to the master record to find the latest checkpoint. If the checkpoint did not complete (e.g. there is no `END CHECKPOINT` record), it can scan back to the previous checkpoint. This method is *fuzzy* because transactions can write records between the begin and end checkpoint records, and these must be considered during recovery.

One additional marker is the FlushedLSN, which points to the last record flushed to disk. The paper suggests also tracking the set of held locks in the fuzzy checkpoint, so that those locks can be restored at recovery time.

1.1.4 Recovery Process (Log Analysis)

1. **Analysis**
   Use the master record to find the last fuzzy checkpoint. From there, replay up to the last committed transaction. There are two types of transactions: winners and losers. The Transaction Table snapshot recovered from the master record reveals which transactions were aborting or running at the time of the snapshot; such transactions are very likely to be losers.

2. **Redo**
   This phase replays (almost) all log entries. It starts by finding the lowest LSN in the Dirty Page Table, which is the earliest transaction that needs to be replayed. The search back in the log is unbounded; consider a very long transaction that had not finished by the time the snapshot was taken. For each log entry, see if it is in the dirty page table. If it is *not* in the dirty page table, it does not need to be applied. If it *is* in the dirty page table, compare the LSN against the LSN recorded on the page; if the LSN on the log entry is older, it does not need to be applied (since it was superseded).

   Note that this step replays both winner and loser transactions.

3. **Undo**
   Next, we must undo the transactions that are marked as losers. Starting from the last record of the latest loser transaction (discovered from the Transaction Table), apply compensating log records (by following PrevLSN pointers). Repeat the process until no loser transactions remain.

   Since each linked list is separate, the undo phase can easily be parallelized, with one processor handling each transaction.

A major point to note is that the idempotent undo/redo actions make multiple crashes (meaning crashes during recover) safe.

Strictly speaking, a separate analysis pass is not necessary, since the redo phase replays both winner and loser transactions.

1.1.5 Backups

The log recovery procedure can be used as a "free" backup system.

1.2 Two-Phase Commit

The two-phase commit protocol is required to support distributed transactions. The paper presents two extensions (Presumed Commit and Presumed Abort, which make assumptions about the outcome of the transaction to reduce communication overhead in the common cases).

The basic idea of the two-phase commit protocol is that any commit must be unanimously accepted by all participants, while any one participant may trigger an abort.

In the following description, any message (in typewriter text) marked with an asterisk indicates that the log is flushed to disk.

The two phase commit begins with a coordinator node sending a `PREPARE` message to all participating nodes (subordinates). Each subordinate responds with a `PREPARE*` or `ABORT*` message. All subordinates wait at a barrier until the coordinator responds.

The two phases begin after this exchange:

Phase 1		
Case:	At least one `ABORT`	All `PREPARE`
Coordinator Action:	Write `ABORT*` record	Write `COMMIT*` record
Phase 2		
Coordinator Response:	Send `ABORT` message	Send `COMMIT` message
Subordinate Action:	Abort and write `ABORT*`	Commit and write `COMMIT*`

After this, all subordinates send an `ACK` message (does not need to be logged). Upon receipt of this from *all* subordinates, the coordinator can forget about the transaction.

In the case of a failure, the position in the protocol is known to all parties, since messages are recorded on durable storage at the critical points; the protocol can be resumed as needed.

In hierarchical versions of the protocol, internal nodes act as coordinators for their transaction sub-trees.

1.2.1 Presumed Abort

The first optimization strategy works on the assumption that transactions that are interrupted due to system failures were in the process of aborting (this means that `ABORT` records never need to be forcibly flushed).

Subordinates that only *read* data do not need to cast a vote since they never make any log records and do not participate in the second phase.

Subordinates casting an `ABORT` vote can immediately forget about the transaction.

Recovery is simple; if the coordinator has no information (since records may have been omitted), it just assumes that the transaction aborted.

1.2.2 Presumed Commit

The idea here is obvious: assume that interrupted transactions were committing and do not force write `COMMIT` messages. This requires an additional forced write from the coordinator `before` all `PREPARE` messages are sent to record which subordinates are participating in the transaction. Let this be called a `COLLECTION` record.

This is required for a slightly edge case: assume a coordinator process crashes before it finishes sending all of the `PREPARE` messages. Some subordinates will be in the `prepared` state, and send a `COMMIT` to the coordinator when they wake up. The problem arises when the coordinator recovers; it forgets about the transaction and it is assaulted with inconsistent (from its point of view) `COMMIT` messages. This way, the recovering coordinator can just send `ABORT` messages to all of the subordinates in its `COLLECTION` record.

`ACK`s are only required for `ABORT`s.

1.2.3 Advantages and Disadvantages of Optimizations

- Under Presumed Abort, readers can drop out of the protocol early.
- Presumed Commit only "wins" if there is at least one subordinate writer.
- Presumed Abort is the most common in implemented systems.

1.2.4 Deadlock Management

Local deadlocks remain a problem and are handled in the standard way. Distributed deadlocks occur when dependencies arise between locks held on different machines as part of concurrent distributed transactions. There are no distributed locks.

The basic method for handling distributed deadlocks is for each node to send *wait_for* messages to the other nodes; a deadlock manager will determine the easiest (measured by the amount of work lost) victim to kill. A DAG is established arbitrarily for the entire cluster, and nodes only send their information "forward" to minimize communication costs. The node which makes the decision to kill something will try to kill local transactions in preference to remote ones to minimize communication costs.

2 Parallel/Distributed Databases

2.1 Parallel

First off, some definitions:

Pipelined Parallelism: This is the basic type of parallelism that is easy to achieve locally; the output of one operator is streamed as the input to the next operator. This was the goal of the System R optimizer.

The second operator can begin working (in some cases) before the first has finished, achieving a mild (but essentially free) speedup.

Serializing operations are a major source of trouble in these cases; one such operation is any sort.

Partitioned Parallelism: The most evident type of parallelism; split data up among several computational nodes and allow them to all proceed in parallel. This is difficult to achieve, of course.

The easiest way to make this efficient (on *database* problems at all amenable to parallelization) is in a shared-nothing architecture, where no nodes can interfere with others.

Speedup: In the linear case, twice as much hardware can perform the same task in half the time.

Scaleup: Again in the linear case, twice as much hardware can perform twice the work in *the same* amount of time. Scaleup is further divided into two classes: batch and transaction. In batch scaleup, the work to be accomplished is presented as a giant blob of work. In transactional scaleup, the work is composed of some number of transactions, and the number of transactions scales up, but the size of each transaction does not (at least, not at the same rate).

The ideals of speedup and scaleup are linear or superlinear (which is much more common in supercomputing, but is also possible on a database workload). Scaleup is, in most senses, the harder to achieve, since it really addresses both time and problem size.

Gray outlines three major "barriers to speed/scale-up" that affect any parallel system:

Startup: The time required to *start* parallel processes; if it takes longer to start the process than to actually do the work, it might not be worth the effort (or is at least questionable).

Interference: Generally introduced by resource contention, arises when each computation node is not completely independent. A common source of contention in a shared-nothing system is network bandwidth.

Skew: Related to the distribution of work among nodes; if data is distributed unevenly, the slowest node determines the speedup of the cluster (in a bad way). This can be considered both in terms of data distribution and request distribution (all requests hit one node, or all data is "partitioned" to one node).

In order to take advantage of partitioned parallelism, data must be partitioned in some way. Some common strategies are:

Round Robin: Works for sequential scans; for any sort of associative operation, it is horribly inefficient. Simple to implement.

Hash: Works for sequential and associative access (especially associative). Difficult to cluster related data.

Range: Good for sequential and associative access, additionally clusters related data. Very susceptible to data skew, however.

It is certainly possible to over-partition and distribute data so much that the increased parallelism gives only higher latency.

2.1.1 Beyond Partitioning Data

While partitioning data is important, relational operators themselves need to be parallelized. One relatively simple way, rather than writing an entirely new set of parallel operator code, is to implement operators in terms of input and output streams. These streams can then handle the splitting and joining of data as appropriate.

Output ports split data to waiting operators, while input ports join data, presenting a single stream to the operator. The input streams can maintain sorted orders while joining with a simple merge operation (if necessary, as in cases where the output of the previous operation was already sorted).

2.1.2 Handling Failures

When a node fails, all of the other nodes need to pick up the workload that the failed node can no longer handle. This assumes that replicas of the data held by the failed node exist.

One method to balance the resulting load is to distributed replicas intelligently: *chain decoupling.* This splits replicas proportionally among all nodes.

R_n is relation n, $r_{n,m}$ is $\frac{1}{n}$ of a replica of R_m, and N_n is node n.

R_0	R_1	R_2	R_3
$r_{3,0}$	$r_{0,0}$	$r_{1,0}$	$r_{2,0}$
N_0	N_1	N_2	N_3

In this example, node n is holds relation n. The replica of relation zero is split approximately evenly among all nodes.

2.1.3 Synchronous Commits

When data replicas must be updated as part of a transaction, the two-phase commit is required.

2.2 Distributed

Problem: cost-based estimation across sites is too difficult (differing setups, data volumes, capacities; entirely incomparable).

Solution: a model derived from economics, wherein participants attach a price they are willing to pay for remote processing of their queries.

The system discussed in the paper was called Mariposa. Key qualities they wanted were to:

- Take into account differing capacities and resources at remote sites.
- Allow completely local and autonomous administrative policies.
- Dynamically balance data (location).
- Have no centralized services.

2.2.1 Bidding Process

The proposed system exists in three tiers: client, middleware, and local execution environment.

1. Client application submits a query with a budget and a bid curve (which describes how the value of the query decreases with time - this affects how quickly the query is scheduled, since there are higher returns for scheduling it early).
2. The parser in the middleware layer parses the query and performs name resolution for the specified relations, along with fragment locations.
3. The *single site optimizer* (also in the middleware layer) performs standard query optimizations, ignoring metadata about locations.

4. The *fragmenter* takes the query plan produced by the single site optimizer and applies location information, producing a fragmented query plan. It then groups related fragments into strides for parallel processing.

5. The *broker* solicits bids from participants in the distributed system, and sends a bid acceptance message to the winner(s).

6. The *coordinator* takes over, and assembles results as they are returned from the winning bidders.

Each server obeys the following protocol:

1. The *bidder* responds to broker requests with bids (prices) on work requests. It takes into account local policies, current load, and available resources when formulating bids.

2. When a bid is accepted, the bidder hands the work off to the *local execution engine*, which performs the actual work and delivers results to the broker.

This bidding process can be expensive (especially for small transactions); small jobs fall back to using *purchase orders*. In this faster system, the host initiating the request sends a purchase order to the site most likely to win a bidding contest. The site either executes the job or passes it on to another host to be executed, and the initiator is simply billed for the service. It is possible that requests made with this abbreviated protocol are not met by their temporal deadlines.

2.2.2 Participant Discovery

The system uses a *directory* to find bidders. Sites with the capacity to service requests *advertise* the data they have and the capacity/bandwidth available in a canonical directory. Advertisements are timestamped to give some indication of how current the information is.

The system, as proposed, also required bandwidth reservations. This is not possible on most modern networks. Bandwidth reservation is required in order to meet deadlines reliably.

2.2.3 Resources

Local disk resources are used to store fragments of relations, which are either generated locally or purchased from other sites (with the expectation that servicing future requests using the purchased fragment will be profitable).

If a fragment gets too "hot" for one site, it might make copies available for purchase. A fragment must be sold to relinquish ownership entirely (if it is the only copy). Similarly, hot relations might be split to relieve pressure.

If a site has multiple fragments of one relation, it can combine them. If a site frequently requests fragments from remote sites, it would probably wish to purchase local copies.

Notably, this system provides two important services: data replication (copies) and data partitioning (fragments). Partitioning, of course, allows for more parallelism in queries (up to a point); fragments are processed in parallel in *strides*.

2.3 Replication

Among the classical replication schemes, there are two axes: *lazy* vs. *eager* and *group* vs. *master*.

Eager Replication: All replicas are updated as part of the updating transaction. Reads on connected nodes are always consistent; reads on disconnected nodes may be out of date. Handling disconnections is difficult in this scenario, for obvious reasons. One common method for addressing disconnected operation is to only allow updates on a *quorum* of nodes.

When disconnected nodes reattach, they are flooded with the updates that they missed.

There are two major problems:

- Disconnected nodes cannot update data.

- The number of deadlocks (failed transactions) is cubic in the number of nodes and transaction size.

Lazy Group Replication: Any node is allowed to update local copies of data. Transactions are sent to all other replicas of the updated data after the transaction commits locally. This implies that concurrent updates can lead to update races, and some conflict management scheme must be present; typically, these schemes use timestamps for reconciliation.

Reconciliation is required whenever a transaction would *wait* on a lock in an eager scheme. A transaction fails (deadlocks) when two transactions wait on each other.

Lazy Master Replication: Each object is *owned* by some node, and only the owner is permitted to make updates. If a node needs to update data which it does not own, it must send an RPC to the owner requesting that the owner execute the transaction on its behalf.

Again, this does not work for disconnected applications. Conflicts are handled as in the eager system: with waits and deadlocks (instead of reconciliation). Somewhat less deadlock-prone than eager group replication, since the transactions are shorter (they do not need to wait for the update to be applied on all nodes).

One important note: for read consistency, read lock requests must be issued via RPC to the owner of any given piece of data.

Non-Transactional Replication: Another type of replication exploits the *convergence* property, to get a type of consistency that is less strict than serializability. While the order of applications is not guaranteed, the end result, after all transactions land, is the same as in a serializable schedule.

Instead of replacing data, at transaction time (e.g. SET x = 5 WHERE ...), values are either incremented, decremented, or replaced (with a timestamp annotation so that order can be resolved). The basis of this type of replication is the idea of *commutative updates*; data transformations that can be freely reordered.

Gray et. al. propose an alternative that maintains consistency, keeps the deadlock rates manageable, and allows for disconnected operation: two-tier replication. There are two main ideas in this scheme:

- Partition the distribution of responsibility to limit conflict points.
- Disconnected updates are *tentative.*

There are two types of nodes: *base nodes* which are always connected and *mobile nodes* which are usually disconnected, but reconnect periodically to synchronize. Mobile nodes maintain two copies of modified data: the last version of data seen from the master node and the tentative version (to which all tentative transactions have been applied).

Transactions on disconnected mobile nodes are *tentative* and may be invalidated upon reconnection if they are unable to be applied in a serializable order. Upon reconnection, tentative transactions are replayed so that they can be converted into base transactions (and therefore produce master data). An *acceptance criterion* is applied to the result of each of these transactions as they are replayed; if the test fails due to a discrepancy between tentative data and master data (say, a bank account balance went negative when applied to master data but not tentative data), the transaction is invalidated.

- When a mobile node is connected, it is essentially operating in a lazy-master system.
- The data shared among base nodes is always the result of a serializable execution.
- As mobile nodes connect, their data converges to the state of the base nodes.
- If all transactions commute, no reconciliations are necessary.

3 Data Access Methods

This section covers a few of what Jignesh classified as "advanced data access" methods. They are indexing methods that are more efficient for some workloads. The first, R-Trees, allow for the indexing of spacial data. The second deals with tricks relating to bitmap indices for some fast aggregate operations with minimal storage wasted on indices.

3.1 R-Trees (Spacial Indices)

R-Trees are used for indexing spacial data. The paper presents its work in two dimensions, but it should generalize to more without too much difficulty. Some common use cases are:

- Finding all objects lying within some region.
- Finding all objects overlapping some region.
- ...

The R-Tree is essentially a B-Tree wherein each non-leaf entry has both a pointer to another node *and* a boundary for the subspace contained in its sub-tree. Leaf nodes point only to data items.

3.1.1 Searching

An R-Tree lookup is parameterized by a search region. At each (non-leaf) node, the search checks to see if a sub-tree overlaps the search region; *all* overlapping sub-trees are searched. At leaf nodes, any entry overlapping the search region becomes part of the result set.

3.1.2 Insertion

Insertion is much like the B-Tree, with node splitting and whatnot if the destination node does not have enough space. The most important element of the algorithm is the policy for choosing in which leaf to place objects. All sub-trees of the current node are considered; the sub-tree chosen is the one for which the insertion of the new object increases the region encompassed by the sub-tree the *least*.

When adjustments are made due to node splitting, bounding regions must be fixed to their new contents. Distribution of objects when splitting is done in a way that attempts to minimize the regions of each new sub-tree. This is a really hard problem, so heuristics are applied.

The splitting algorithm is critical for good packing and performance. The authors examine several, including: exhaustive, quadratic, and a linear version. They conclude that the linear method is the best, but their own numbers show better results on lookup and storage utilization for the quadratic method. Perhaps CPU time was more important to them at the time.

Quadratic:

1. Pick the two nodes (from the $M+1$ total nodes) that would waste the most space if placed in the same group, and take them as seeds for the two new nodes.
2. For each of the remaining nodes n, calculate the required region expansion for adding n to each group (d_1 and d_2, respectively). The next node to be inserted is the one which has the greatest difference between d_1 and d_2. This essentially selects the node with the greatest preference for a region.
3. Add the node from the previous step to the node it prefers. Go to 2 until done.

Linear: The linear method is simpler, of course, and makes compromises. The next object to insert is chosen arbitrarily, and the major difference is in how the seeds for the two sets are chosen.

1. For each dimension, find the region which is most extreme (the largest) in that dimension.
2. Normalize all of the resulting regions.
3. Choose the two with the greatest (normalized) separation along any of the dimensions.

A problem with splitting is that the last $\frac{m}{2}$ nodes can all be dumped into one group, which may be far from optimal.

3.1.3 Condensing Nodes

Instead of implementing a full condense operation, which would be somewhat hard, an easy solution is to simply remove the nodes to be condensed and re-insert them. This can even lead to better distributions, since the region may not need to be re-created and the objects could end up in better fitting regions.

Another important reason for choosing the simple strategy is that the notion of "closeness" is underdefined for an R-Tree, whereas it is simple in a B-Tree.

Updates proceed similarly, as changes to bounding boxes are simply difficult to propagate.

3.1.4 Bulk Loading Data

Bulk loading data in a B-Tree is simple; sort the data and take that structure as the list of leaves. After that, simply build up a B-Tree structure on top of this list. There is no single ordering for spacial data, so the problem is harder.

One solution is to linearize the data. There are two common methods: Z-Value sorting and a Hilbert space sort. With such a sort applied to the objects, build an R-Tree structure on top of it. The resulting trees can often exhibit better packing than dynamically built trees.

3.1.5 Variant: R* Tree

This craziness doesn't automatically split nodes when they fill up. Instead, it just removes some (around thirty percent) and re-inserts those. Supposedly, this gives better packing (of course, that depends on the heuristics used to choose the nodes to be re-inserted).

3.2 Bitmap Indices

The new indices introduced in this paper are not trivial to update, and so are intended for data warehousing workloads where updates are infrequent and offline.

3.2.1 Traditional Indices

Bitmap: Each row in the indexed relation is represented by one bit in the bitmap; conceptually, if row i has the property represented by the bitmap, bit i is set. In practice, the most likely mapping is to use the page number and offset into the page of a given row as its index into the bitmap (since this is a more stable property). One suggested method to facilitate this method is to place a fixed limit on the number of tuples in any given page; the entries that are actually used in a page are tracked as metadata.

Most useful attributes would require more than one bitmap index. A significant space savings can be realized by using run length encoding on the bitmaps. Many useful properties (including some aggregates) can be computer with simple bit operations.

Value-List: This is another name for the standard B-Tree index. Leaves point to data items and the B-Tree provides a lookup by the indexed attribute. Leaf nodes can be bitmaps, so the two concepts can be combined.

3.2.2 Projection Indices

The projection index stores the *values* of one attribute in a relation in order. Holes in the row identifiers are preserved. For fixed sized data, this is very useful; recovering the real location (page and slot number) are trivial once an "interesting" value is found.

3.2.3 Bit-sliced Indices

Instead of storing one bitmap for each possible value on an attribute, the bit-sliced index slices along bit boundaries; as an example, instead of requiring 2^8 bitmaps to index a byte column, only 8 bit-slices would be required[1].

To compute range queries, use three temporary bit vectors: $B_<$, $B_>$, and $B_=$. Slide each of the three bitmasks from left-to-right across the bit slices, setting bits to be "sticky" as appropriate for each of the vectors.

3.2.4 Computing Aggregates

todo

3.2.5 Join Indices

Consider a join $\sigma B \bowtie A$ which occurs often on a dataset. Instead of building an index on each of A and B and using them to perform the join, build an index on the *result* of the join. That is, build the index on the values of B, but have the index map to the values of A.

With this sort of index available, the actual join only needs to be computed at index creation time.

4 Data Models

These are a few alternative data models for those who feel that they are too good for relational algebra.

4.1 Object Oriented Databases

The idea behind this class of systems is to essentially support persistent objects in the host language (often C++). The resulting systems are typically set-oriented, and queries take the form of *path* expressions, which describe relationships by the linkage between different types of objects. Even so, most of the systems exhibit significant scaling problems, both in terms of data set sizes and in concurrency (which is largely ignored).

Many of the systems involved language extensions that relied on vendor-supplied compilers (that is never a good thing). An additional problem involves language interoperability, which is exceedingly difficult in this type of system (since objects have different representations in every language).

ObjectStore makes a distinction (which does not affect the user very much) between persistent objects (backed by the database) and transient objects (which are standard objects in the host language). Transactions are supported. Appropriate container types for the results of set-valued operations are chosen by the system as appropriate. Relations between objects are maintained *bi-directionally* automatically. Where possible, queries are constructed at compile time.

Two important implementation techniques make this (or at least ObjectStore) possible: virtual memory tricks and bit swizzling.

4.1.1 Abusing Virtual Memory

Objects are stored in pages on persistent storage in the same format as they would be represented in-memory in their host language. To load them into memory for the active process, the object database system simply memory maps the entire page into memory (it could also pull in surrounding pages if the clustering was correct and it anticipated a need for them soon).

`mmap` is an expensive operation, but the cost is amortized over the number of objects brought into memory at once. Writes to this memory can directly update the on-disk representation of the objects.

One very important complication in this setup is the pointers held by the objects brought into memory. When on disk, these pointers refer to other objects via some logical mapping, possibly disk offsets. When `mmap`ed, these pointers are nearly meaningless; the "solution" is to *swizzle* them, replacing the logical pointers

[1]NULL values are typically tracked with one extra bitmap.

with actual memory pointers. A tag table is maintained to determine which pointers have been modified, and which have not.

Obviously, care must be taken to re-swizzle them into logical forms when persisting them to disk (and application crashes could really be problematic).

The system relies on native virtual memory hardware for most of the heavy lifting (catching traps, cache management, etc). Pages can be shared in read-only or exclusive modes, and notifications of lock ownership changes are handled via AFS-style callbacks.

Virtual memory space limits on 32 bit systems can be problematic; to address this limitation, the system can use virtual memory "windows" into regions of the database actively being used.

4.1.2 A Few More Drawbacks

- Indices are very hard to maintain
- Join optimization is even harder
- Relations are not known by name
- No schema
- Runs in the same address space as applications (bad on all counts), a significant point from Stonebraker.

A compromise solution is the (now common) Object-Relational Database.

4.2 Object Relational Databases

An object relational database is an incremental improvement to a typical relational database that gives many of the benefits of full object-oriented databases with fewer drawbacks.

The primary offerings of object-relational database systems are:

- User-defined datatypes (which can be used as standard attributes)
- User-defined functions for using these datatypes
- User-defined access methods
- (Sometimes) User-defined operators for query predicates (such as "contains" for some sort of spacial type)

These types of facilities are often offered as add-on packages to databases by vendors themselves. One problem with this model (depending on perspective) is that query optimizers are not aware of the characteristics of user-defined types; this can lead to marked inefficiencies in comparison to standard data types, since the optimizer must make conservative assumptions.

Most ORDBMS extensions are not standardized in any way, which can make interoperability interesting (in the Discworld sense).

4.3 XML/Ad-Hoc Data Models

Some people consider the strict structure imposed by relational databases to be problematic; in response, some data models offer "schema-later" or schema-less models. One such popular data model is XML, in which a schema is optional[2] or may be imposed at a later time, after data is done evolving.

In these systems, data is supposed to be "self-describing" so that it can be processed by any system without a schema. This never works in practice (at any sort of appreciable scale).

Queries can be run against XML documents using XQuery (and the FLOWR language) or XPath (for simpler queries). Both of these query methods are set-at-a-time models. The overall data model is much more complicated than a relational database system (this complexity is reflected in both the specification and they query languages). Graphs are allowed (via node references), as are union types.

[2]The schema may be optional, but complex XML documents with no schema are essentially useless.

5 Data Mining

5.1 Association Rule Mining

Association rule mining is a database-flavored application of machine learning techniques; in particular, it is interested in finding correlations between transactions. Correlations are recognized in terms of minimum support with a certain confidence (the thresholds are, of course, highly application dependent).

Let T be the set of all transactions, and let T_1 and T_2 be the sets of transactions concerning items 1 and 2, respectively. Together, T_1 and T_2 form an itemset (which can either be significant or not, depending on the thresholds set).

Support $= \frac{N(T_1+T_2)}{N(T)}$. The support is the fraction of total transactions which contain the itemset under consideration.

Confidence $= \frac{N(T_1+T_2)}{N(T_1)}$. Confidence is a measure of how significant the relationship between T_1 and T_2 is. Consider the case where $N(T_2) \gg N(T_1)$; the confidence will be very low, since T_1 contributes almost nothing to the overall set of transactions, but the itemset of items 1 and 2 still has significant support (due to T_2).

There are several algorithms for finding association rules, but this paper introduced the Apriori-family. This entire class of algorithms exploits the property that is their namesake: *If A is a frequent item set, then any subset of A is also frequent.*

Roughly, this set of algorithms can be broken down into two steps:

1. **Find frequent item sets (based on support)**
   Start by collecting all frequent item sets of size 1. This means all items that are present in the full set of transactions with greater than the minimum-support.

   Now iteratively construct sets of size i by extending sets of size $i-1$. This directly leverages the a-priori property through dynamic programming. At each step, prune impossible sets (that is, sets which have subsets which are *not* frequent, since these violate the a-priori property).

2. **Find rules (based on confidence)**
   Given the sets of $Y \rightarrow Z$ association rules discovered above, simply compute the confidence for every frequent item set, discarding those with confidence less than the specified minimum. If possible, computations of $N(Y \cup Z)$ should be preserved from phase 1 to be reused.

Finally, the most time consuming and subjective part of the process is to prune "uninteresting" rules, however that is defined for the application domain.

5.2 OLAP

The Data Cube paper covers a wide range of topics, from its main point regarding generalized group-by queries to sundry good ideas and implementation techniques.

5.2.1 Data Warehousing

Data enters a warehousing environment through some kind of On-line Transaction Processing (OLTP) system; typically, this data comes from some kind of real-time operation (e.g. a sales floor). This data resides in multiple databases, possibly with duplicate data and minimal indexing.

Before it can be analyzed, the data typically must be cleaned and normalized through various (domain-dependent) transformations. This is the warehoused form of the data (which is updated periodically, but not in real-time).

From there, it is typically subject to one of two fates:

- Data mining (through machine learning-style methods, possibly tailored to massive non-memory resident data sets)
- On-Line Analytics Processing (OLAP). Enter the Data Cube.

5.2.2 The Data Cube Operator

The idea of the data cube is to generalize the group-by operator to work along many (d) dimensions at once without having to write many separate queries (which cannot share data). There are actually two new operators (the first is general, the second is more specific and can avoid some computation):

Cube: Computes all 2^d possible group-by combinations, forming a d-dimensional hypercube of data aggregations. Dropping one dimension of data (reading along an edge of the hypercube) gives a *more general* aggregate view of the data.

Rollup: The rollup operator is a special case, typical in reporting contexts, where not all of the summary views are required; it simply does not compute the unneeded elements.

One very important question arises when computing these aggregates: what value is placed in entries representing aggregated data? Gray introduced the `ALL` pseudo-value, which behaves much like `NULL`, except it is set-valued and takes a different value depending on context (that value is the set of all values being combined in the aggregate). This is a semantically light-weight issue, but is an elegant way to deal with the requisite schema mismatch (and allows all of the results to be reported as a single relation, rather than many relations of varying sizes/schemas). Fields which have no relevant value at a given dimension are filled with this `ALL` value.

Computation of the data cube has two steps: (1) compute the core hypercube (essentially, all of the most specific group-bys) and then (2) computing combined aggregates by "drilling down" one dimension in each direction (recursively). There are several types of aggregate to consider when computing the second phase of the data cube:

Distributive: This type of aggregate requires only a single aggregation variable, independent of the cardinality of the data being aggregated. The core hypercube can be decomposed via projection, and the aggregates computed based on these projections can themselves be composed.

Algebraic: A constant number (greater than one) of aggregation variables are required. A similar projection scheme can be used, but the intermediate handles to aggregation iterators must be preserved (since their internal states are required for further composition – just final results based on their aggregations is not enough. Consider the average aggregate).

Holistic: No upper bound exists on the storage required by the aggregate computation. There are no short-cuts known, and the essentially brute-force 2^n algorithm (which actually performs $T2^d$ computations) must be used.

It is important to consider cardinalities of relations along each edge of the core hypercube when performing these decompositions; projecting along one axis may offer significant computational savings over simply choosing one at random.

5.2.3 Nice Stuff

Gray mentions a few (almost) common sense extensions to SQL that are useful for the data cube operator, but are generally useful for anyone (and are implemented by many vendors):

- Functions in group by clauses.
- Functionally dependent fields in select lists with aggregate operators