ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract Data Management Systems

Roman Böhringer

February 13, 2021

Department of Computer Science, ETH Zürich

Contents

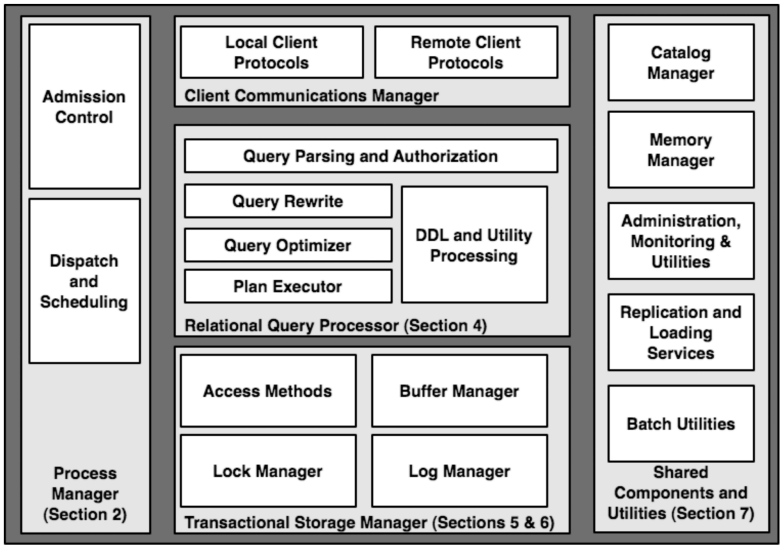
Chapter 1

Intro & Basics

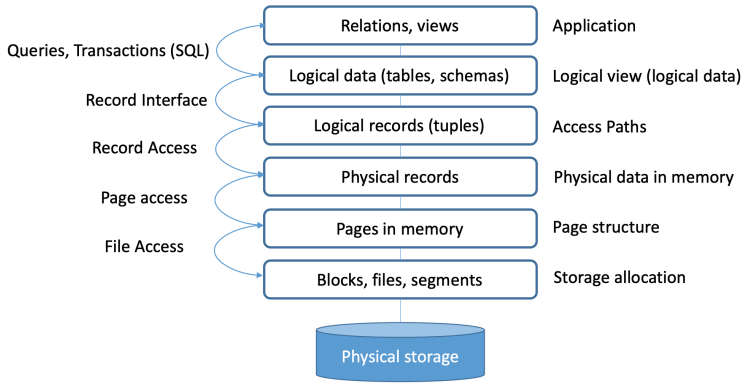
DBMS (database management systems) are used to avoid redundancies/inconsistencies, to provide rich and synchronized (concurrent) access to the data including recovery after system failures and security/privacy, and to facilitate reuse of the data.

A key concept of databases is data independence. We distinguish between physical data independence (the storage format/representation in memory is hidden, the user sees just relations and a schema) and logical data independence (views over the schema can be built to allow different logical interpretations of the same data).

The main components of a DBMS are:



Another view on the architecture involves the different abstraction levels and their access methods/interfaces:



1.1 Process Models

A DBMS worker is the thread of execution in the DBMS that does work on behalf of a DBMS client (software component that implements the API used by application programs). There are three natural process model options:

- **Process per DBMS worker**: Relatively easy to implement and provides isolation, but shared data structures like the lock table, buffer pool, or log tail need to be allocated explicitly in shared memory. Furthermore, there are scaling issues because a process has more state than a thread, context switching is slower, and implementing parallel query processing is more involved.
- **Thread per DBMS worker**: Scales well to large numbers of concurrent connections, but portability, debugging, and isolation can be an issue.
- **Process/Thread Pool**: There is a pool of processes/threads (with a bounded and often fixed size) and a request is given to one of them. Parallel query processing is easier to implement and the approach scales better.

Threads can be kernel-level threads or DBMS threads that are scheduled by a DBMS process (user-level threads). The DBMS threads can be scheduled on OS processes or OS threads (e.g. SQL Server Fibers). Besides process pools, the DBMS workers can also be multiplexed over a thread pool. PostgreSQL has a postmaster (daemon) that spawns server processes for clients.

Admission control can be done in the dispatcher process (ensuring that the number of connections is kept below a threshold) and in the query processor (more sophisticated decisions based on expected memory usage, CPU load, or I/O pattern).

We distinguish between client processes and server process (which both can be implemented as threads or processes) where a client process is the execution unit that runs on the client (and is used to connect to the database) and the server process the unit that runs inside the database (used to execute queries on behalf of a client).

1.2 Parallel Architecture

We distinguish between shared-memory (all processors can access the same RAM/disk with the same performance), shared-nothing (cluster of independent machines that communicate over a network interconnect), shared-disk (e.g. with SANs), and NUMA (nun-uniform memory access, different memory access latencies) systems. For shared-memory and NUMA (with some locality optimizations) systems, the process model is very similar to the uniprocessor approach. There still can be data or function shipping, but programming is much easier because of the shared memory abstraction. In shared-nothing systems, horizontal data partitioning where each tuple is assigned to an individual machine is often used. Partial failure has to be considered where the system can either stop processing queries, skip the missing tuples ("data skip"), or use redundancy schemes to cope with these issues. Shared-disk systems can share data over the disk, but require a distributed lock manager facility and a cache-coherency protocol. They separate storage from compute.

Generally, function shipping (the query goes to where the data is) and data shipping (the data goes to where the query is) is used when executing queries, depending on the architecture (for shared disk only data shipping, for the other architectures usually a combination depending on the query).

1.3 Relational Query Processor

The database operators are:

- σ Selection
- π Projection
- $\times$ Cartesian Product

- ρ Rename
- $\cup$ Union
- $-$ Set Minus
- $\bowtie$ Join
- $\div$ Relational Division
- $\cap$ Intersection
- $\ltimes$ Left Semi Join (natural join with only attributes of the left relation)
- $\rtimes$ Right Semi Join
- $\mathbin{⟕}$ Left Outer Join
- $\mathbin{⟖}$ Right Outer Join

All other operators can be derived from the first six operators.

A relational query processor takes a declarative SQL statement, validates it, optimizes it into a procedural dataflow execution plan, and (subject to admission control) executes that dataflow program on behalf of a client program.

The query rewriter simplifies and normalizes the query without changing the semantics. It expands views, evaluates arithmetic expressions, can rewrite predicates (better utilization of indices, satisfiability tests, using transitivity to introduce new predicates), performs semantic optimization (e.g. redundant join elimination when the result of a join is never used and does not change the number of tuples because of integrity constraints), can flatten subqueries, and perform other heuristic rewrites.

The query optimizer's job is to transform an internal query representation into an efficient query plan for executing the query. A query plan (represented as machine code or interpretable) can be thought of as a dataflow diagram that pipes table data through a graph of query operators. There are many things to consider when designing a query optimizer, among others the plan space (which plans to consider), the selectivity estimation (with different techniques to estimate the size of selections), search algorithms, and parallelism.

Query executors often employ the iterator model where an iterator can have multiple other iterators as inputs and provides a `get_next()` function that returns a (pointer to a) tuple. For parallel query execution, there are special exchange iterators. The iterators typically place tuple descriptors (arrays of column references; references to tuples including column

offsets) in memory. The tuples themselves either reside in the buffer pool (BP-tuples; pin count of the page must be incremented/decremented accordingly) or on the memory heap (M-tuple; either copied from the buffer pool or the result of expressions). There are tradeoffs between BP- and M-tuples and DBMS often use both types. For data modification statements, the "Halloween problem" needs to be avoided. This refers to a phenomenon in which an update operation causes a change in the physical location of a row (in the index), potentially allowing the row to be visited more than once during the operation.

Access methods are the routines that manage access to the various disk-based data structures that the system supports. These typically included unordered files (“heaps”), and various kinds of indexes. All major commercial systems implement heaps and B+-tree indexes. An access method provides an iterator API with an `init()` method that accepts a "search argument" (SARG). With the SARG, only the tuples that match the criteria are returned by the access method and unnecessary tuple copies are avoided.

Data warehouses require specialized query processing support like bitmap indices, fast load (fast bulk-loading of data), materialized views (with issues such as which views to materialize, maintaining the freshness of the views, and usage of the materialized views in queries), providing optimized aggregate queries (data cubes), and the optimization of snowflake schema queries.

Modern DBMS are extensible in various ways. They can provide extensible data types (abstract data types), support structured/semi-structured types such as arrays, sets, trees, or XML, and provide full-text search with inverted indices.

1.4 Storage Management

The DBMS can either directly interact with the low-level block-mode device drivers or use standard OS file system facilities (which is more common today). Direct access provides better spatial (DBMS can optimize data consecutively) and temporal (DBMS can control when data is flushed and control prefetching, whereas both is controlled by the OS when a filesystem is used) control. But both problems are not very prevalent today as the file system typically places close offsets in a file in nearby physical locations and modern operating systems provide facilities to bypass disk buffers.

There are also specialized page replacement schemes for databases that include information from the query plan.

1.5 Transactions

ACID is a set of properties of database transactions intended to guarantee data validity despite errors, power failures, and other mishaps:

- **Atomicity**: A transaction is executed in its entirety or not at all.
- **Consistency**: A transaction executed in its entirety over a consistent DB produces a consistent DB. Consistent means that the database state is the result of correctly applying operations to the database (where the underlying model of correct operation execution is sequential execution).
- **Isolation**: A transaction executes as if it were alone in the system.
- **Durability**: Committed changes of a transaction are never lost (can be recovered).

We assume that transactions are correct, i.e. the user knows what is correct. But there are mechanisms to prevent incorrect data modifications (by aborting the transaction) like consistency constraints on tables, constraints on values, referential integrity, or triggers applying consistency checks. An implication of atomicity is that undo or redo has to be performed for non-completed transactions. As consistency is only given for completely executed transactions, isolation is required to accomplish it (when transactions can see the intermediate state, their input and therefore their output is not guaranteed to be consistent). Isolation implies that we need to be able to determine when a transaction is done and detect/resolve conflicts. It is enforced through concurrency control mechanisms (locking). For durability, the log and snapshots allow to go back and forward in the history of the database by undoing/redoing transactions.

The main techniques of concurrency control enforcement are:

1. **Strict two-phase locking (2PL)**: Transactions acquire a shared lock on every data record before reading it, and an exclusive lock on every data item before writing it. All locks are held until the end of the transaction, at which time they are all released atomically. A transaction blocks on a wait-queue while waiting to acquire a lock.
2. **Multi-Version Concurrency Control (MVCC)**: Transactions do not hold locks, but instead are guaranteed a consistent view of the database

state at some time in the past, even if rows have changed since that fixed point in time.

3. **Optimistic Concurrency Control (OCC)**: Multiple transactions are allowed to read and update an item without blocking. Instead, transactions maintain histories of their reads and writes, and before committing a transaction checks history for isolation conflicts that may have occurred; if any are found, one of the conflicting transactions is rolled back.

There are locks and latches. Locks are kept in a lock table and located via hash tables whereas latches reside in memory near the resources they protect. Locks are subject to the strict 2PL protocol whereas latches may be acquired or dropped during a transaction based on special-case internal logic. Latching coordinates threads to protect in-memory data structures, locking coordinates transactions to protect database contents. Deadlock detection and resolution is usually provided for transactions and locks but not for threads and latches. Deadlock avoidance for latches requires coding discipline and latch acquisition that fails rather than wait.

SQL provides the following isolation levels:

1. **READ UNCOMMITTED**: A transaction may read any version of data, committed or not. This is achieved in a locking implementation by read requests proceeding without acquiring any locks.
2. **READ COMMITTED**: A transaction may read any committed version of data. Repeated reads of an object may result in different (committed) versions. This is achieved by read requests acquiring a read lock before accessing an object, and unlocking it immediately after access.
3. **REPEATABLE READ**: A transaction will read only one version of committed data; once the transaction reads an object, it will always read the same version of that object. This is achieved by read requests acquiring a read lock before accessing an object, and holding the lock until end-of-transaction.
4. **SERIALIZABLE**: Full serializable access is guaranteed.

Because of the phantom problem, REPEATABLE READ and SERIALIZABLE are not identical. On an aggregation query (e.g. `SELECT COUNT(*)`), different results are allowed for REPEATABLE READ when new tuples are inserted between two calls.

The main components of the transactional storage manager are:

1. **Lock manager (concurrency control)**: Uses a deadlock detector thread that examines the lock table to detect waits-for cycles and aborts transactions.
2. **Log manager (recovery)**: Often uses the write-ahead logging protocol with three rules:

   a) Each modification to a database page should generate a log record, and the log record must be flushed to the log device before the database page is flushed.

   b) Database log records must be flushed in order; log record r cannot be flushed until all log records preceding r are flushed.

   c) Upon a transaction commit request, a commit log record must be flushed to the log device before the commit request returns successfully.

   Recovery does not need to start at the very first log record, but at the log record describing the earliest change to the oldest dirty page or the log record representing the start of the oldest transaction (whichever is earlier; this is called the recovery log sequence number/recovery LSN). Computing the recovery LSN is called checkpointing.
3. **Buffer pool (staging database I/Os)**
4. **Access methods (organizing data on disk)**

1.6 Shared Components

The database catalog holds information about data in the system (users, tables, schemas, columns, indices, etc...) and is a form of metadata. It is itself stored as a set of tables in the database.

DBMS usually use a context-based memory allocator. A memory context is an in-memory data structure that maintains a list of regions of contiguous virtual memory. Contexts are often allocated, used by a part of the DBMS (e.g. the query optimizer), and then completely deleted which can improve memory allocations (as it does not result in many `malloc()`/`free()` calls/allows optimizations based on memory usage patterns).

For ideal performance, disk management subsystems need to handle device-specific details like the performance characteristics of RAID systems.

For replication, there are three typical schemes. The entire database can be duplicated every replication period (physical replication), special replication tables that are replayed at distance locations can be used (trigger-based replication), or log writes can be intercepted and delivered to the remote system (log-based replication).

Chapter 2

Storage

A big part of the performance of databases arise from proper storage management, adequate data representations, and suitable optimizations on organizing the data in memory. Databases exercise the whole storage hierarchy (CPU registers, caches, main memory, local persistent storage, remote persistent storage, archive storage) and need to deal with the huge capacity and latency gap between different levels. Furthermore, the access methods differ. Caches/DRAM provide byte addressable random access whereas the lower layers provide block addressable sequential access. To improve performance, it is important to exploit spatial and temporal locality, hide latency with pre-fetching, have smart caching/replacement strategies, and write modifications back only when needed.

In cloud environments, the problem is different as there is a clear separation between the compute and storage layer (connected over the network).

2.1 Segments and File Storage

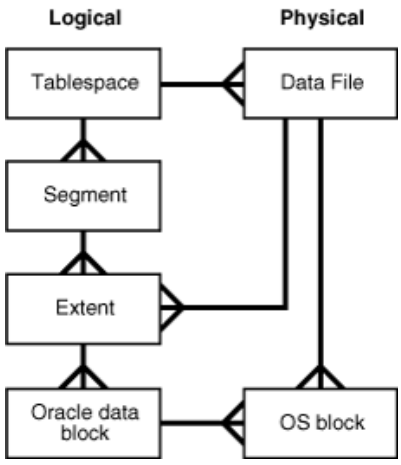
Logically, there are:

- **Tablespaces**: A logical data unit in the database for schema related (tables, indices, clustered tables) or engine related (result buffers, undo buffers, etc...) data. Data is not necessarily contiguous, but space (memory/disk) is allocated to tablespaces.
- **Segments**: A segment is a collection of extents. One tablespace is made up of one or more (e.g. with partitioning) segments.
- **Extents**: An extent is a collection of contiguous blocks. It is allocated/dropped as a unit. When a segment is created, it is allocated an extent (with a tunable initial size, where large extents offer

maximal performance because of contiguous data, but possibly introduce fragmentation when they are not full). When more space is needed, another extent is created (typically larger than the previous one, e.g. exponentially by a factor of 1.25 which bounds the size of the extent directory). Extents therefore are a compromise between a static file mapping (very efficient, but no flexibility) and dynamic block mapping (maximum flexibility, but non contiguous and therefore with poor performance).

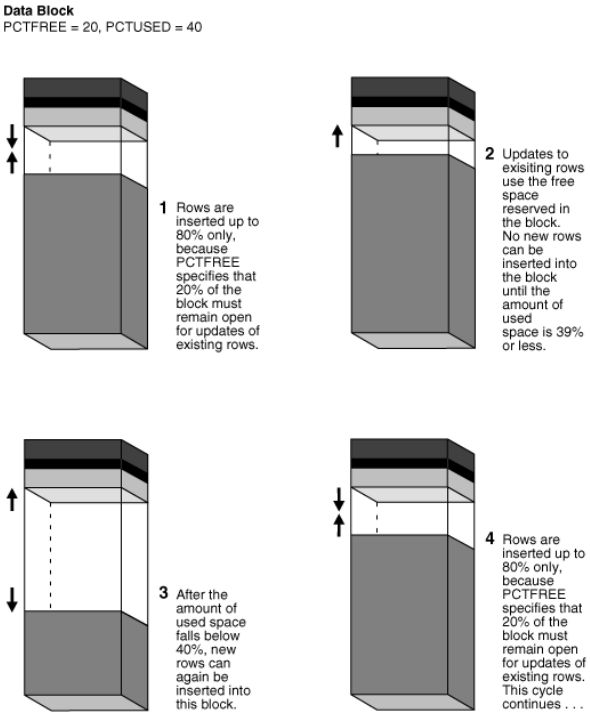
- **Blocks**: Blocks are similar to OS pages, but not necessarily the same size (often larger). Allocating space page by page would be too much overhead and larger blocks can improve spatial locality.

Physically, we only have data files and OS blocks. The relationship between these concepts is illustrated in the following diagram:



Blocks are structured as slotted pages. There is a header (containing metadata), potentially a table directory (with the schema of the table, can also be in a separate catalogue), and a row directory with pointers to the actual tuples stored in the block. The row directory grows downwards, wheres the space for tuples is used upwards (or the other way around). The row directory allows to insert tuples anywhere in the block (and change their position) with simple references to the tuples by the block and slot number (record id that does not change). For optimizing the use of blocks, there is the concept of "percentage free" and "percentage used". The "percentage free" value (e.g. 20%) determines how much space in each block is reserved for updating instead of storing new tuples (because tuples can get bigger after updates). "Percentage used" (e.g. 40%) determines the threshold of used space after which new rows can

be inserted again into the block. This avoids constant updating of the free list.



Compaction within blocks is usually only done (because it is expensive) when the block has enough space for an operation but the space is not contiguous. When the new size after an update does not fit in the block, the original space is used to store a pointer to the new row.

Each segment contains one or more free (or used) lists, containing pointers to blocks that have usable free space (and potentially the amount of free space). The list can be a bottleneck, therefore several can be used, the traversal can be made fast by keeping it small (larger pages, sorting by available size, caching positions), and it should be possible to find holes efficiently (store locations of space in each page approximately, etc...).

For writing to disk, shadow paging can be used instead of in-place updates: On an update, a page is copied (in the same extent) and the update is performed on the copy. When the transaction commits, the old page is marked as free and the copied one marked as active page. This makes recovery easy and can be good for concurrency control (snapshot isolation), but page access and management is complicated and updates can destroy locality. An alternative are delta files where pages are copied to a

separate file before modifications. The update is either performed on the original page or on the copy in the delta file.

2.2 Database Buffer Cache

Databases cache blocks in memory, writing them back to storage when dirty (modified) or in need of more space. The buffer cache generally consists of latches, hash buckets, and a linked list of buffer headers. The latches avoid conflicting access to the hash buckets and generally cover several hash buckets (otherwise, too much space would be needed for them). Contention on the latches may cause performance problems (e.g. for hot blocks). This can be addressed by reducing the amount of data in a block, using more latches, or using multiple buffer pools. To find the correct linked list where a block header resides a block identifier (e.g. the file ID and block number) is hashed. The linked lists contain the block headers (with various information such as block number, type, format, log sequence number, replacement information, etc...) and should be kept short. Besides normal blocks, there can also be version blocks (copies done for shadow paging, older versions can be found but linked list will grow too long with many updates), undo/redo blocks (for recovery), dirty blocks, pinned blocks, and more.

Buffer replacement for the buffer pool is similar to the OS, but the database has much more information on how and when data will be used. LRU can perform poorly for databases because of table scan flooding (large table loaded to be scanned once) or index range scans (range scan using an index) that pollute the cache with pages that are not needed anymore. A way to avoid this is to put blocks that are rarely accessed at the bottom of the list or to simply not cache large tables. We generally want to evict clean pages as they do not need to be written to storage. Second chance approximates LRU. The buffer is treated as a circular buffer and each page has a counter that is set to 1 on access. When the eviction passes by, it sets the counter to 0 if it was 1 or evicts the page if it already was 0. Clock sweep is the same as second chance but keeps a counter instead of a 1/0 flag (therefore taking the access frequency into account). In 2Q, two lists are used. A FIFO list for blocks that do not need to be kept and a LRU list for blocks that are accessed several times. Blocks are migrated between the two lists and eviction is done from the FIFO list. Oracle uses a keep buffer pool (separate buffer for pages that should not be evicted) and a recycle buffer pool (for pages that should not be kept). Furthermore, they use touch count (hot/cold list) which is a more sophisticated LRU. A count of accesses is kept and frequently accessed pages float to the top

(hot), whereas rarely accessed pages float to the bottom (cold). Counters are periodically decreased and to avoid counting problems (when a page is accessed many times, but only for a short interval), the counter is incremented only after a certain number of seconds. Postgres uses a ring buffer for scans and allocates pages in a ring, evicting the pages from the beginning of the ring when it is full.

2.2.1 Advanced Algorithms

Besides conventional virtual memory page replacement algorithms like least recently used (LRU), MRU, FIFO, or CLOCK, there are buffer management strategies that exploit the knowledge of the access pattern.

The domain separation algorithm classifies pages (statically) into types, each of which is separately managed in its associated domain of buffers (by LRU). The "new" algorithm divides the buffer pool on a per-relation basis and orders the relations by priority. Within each relation, MRU is used. The set of pages over which there is a looping behaviour is called a hot set. The hot set algorithm allocates buffers according to the size of the hot set of a query, a new query is allowed to enter the system if its hot set size does not exceed the available buffer space.

The DBMIN algorithm uses the query locality set model (QLSM). Buffers are allocated on a per file instance basis and each file instance is given a local buffer pool to hold its locality set, the set of the buffered pages associated with the file instance. This set will vary depending on how the file is being accessed, which DBMIN takes advantage of. Furthermore, the buffer pool associated with each file instance is managed by a replacement policy that is tuned to how the file is being accessed.

To evaluate buffer management algorithms, we can use direct measurements, analytical modelling, or simulation. For simulation, there is trace-driven simulation (with traces recorded from a real system) and distribution-driven simulation (where events are sampled from a distribution).

2.2.2 Query Access Patterns

In the query locality set model, we differentiate the following query access patterns:

- Sequential:
  - Straight sequential (SS): Sequential scan (of unordered tuples) without repetition

  - Clustered sequential (CS): Sequential scan with local rescans (e.g. in merge join with non-unique keys)

  - Looping sequential (LS): Several repetitions of a sequential reference (e.g. in nested loop joins)

- Random:

  - Independent random (IR): Series of independent accesses (e.g. index scan on a non-clustered index)

  - Clustered random (CR): Series of independent accesses with local clusters (e.g. when the inner relation of a join is a file with a non-clustered and non-unique index)

- Hierarchical:

  - Straight hierarchical (SH): Sequence of page accesses that form a traversal path from the root down to the leaves of an index with only one index traversal.

  - Hierarchical with straight sequential (H/SS): Index traversal, followed by a straight sequential scan.

  - Hierarchical with clustered sequential (H/CS): Index traversal, followed by a clustered sequential scan.

  - Looping hierarchical (LH): Repeated accesses to the index structure (e.g. when the inner relation of a join is indexed on the join field)

2.3 Case Studies

2.3.1 SAP HANA NSE

HANA's Native Store Extension (NSE) provides a unified persistent format and pageable primitives (multi-page vectors, paged mapped vectors, arbitrary sized items) to allow hybrid columns. Compressed pageable columns (with run-length encoding, sparse encoding, and indirect encoding) are built upon the primitives with additional helper structures (in-memory or on disk) to improve performance (e.g. allowing binary search on the pages by keeping the ranges of a page in-memory). The buffer cache uses an adaptive version of LRU that allows domain-specific page pools and keeps hot buffers in memory. The cache allows asynchronous prefetching

of pages. The cache is integrated into the execution engine which e.g. sends the prefetch requests if they are sensible or provides other hints. Columns, partitions, and tables can be tagged with load unit hints which change the preferred storage format (in-memory or paged) and there is a load unit advisor that uses heuristics based on the access statistics to give recommendations for the load unit (with the goal of saving memory or improving performance).

2.3.2 Storage in Cloud Environments/Snowflake

Amazon's simple storage service (S3) is an object storage service in the cloud. There are no in-place updates (objects must be written in full), but parts (ranges) of an object can be read instead of the whole object.

Snowflake is a data warehousing solution (specialized for analytical queries that may take hours) for the cloud that separates storage (Amazon S3) and compute (virtual warehouses, i.e. collection of worker nodes, for query execution, cloud services for management). Tables are horizontally partitioned into large, immutable files (micro-partitions) and write operations produce a newer version of the file (which also allows snapshot isolation with MVCC and time travel). Data in the micro-partitions is stored in columnar form. Queries only need to download the file headers and the columns they are interested in, metadata is kept in a distributed key-value store. Each worker node maintains a disk cache of table data with a simple LRU policy, there is no buffer pool (but operators can spill to disk if memory is too small for the execution). Min-max based pruning (also known as materialized aggregates, zone maps, or data skipping) is used instead of indices: Data distribution information (in particular the minimum/maximum value within a chunk) is used to skip whole chunks during execution. It also uses dynamic pruning during execution (when e.g. additional information about the needed value ranges become available in a join). Snowflake supports semi-structured data and performs automatic schema discovery/type inference and automatically extracts common paths into columns. Furthermore, it performs optimistic data conversion (storing the converted and original values).

Chapter 3

Access Methods

A tuple usually contains a header (validity flags for deletion, visibility info for concurrency control, null bit maps) and the actual attributes. For performance reasons, variable length data is often placed at the end and the beginning of a tuple contains the lengths and offsets for variable length data (to avoid the linear access time when only the length is stored). In contrast to schema-less systems, schema information is not stored in the tuple in relational engines. For very large tuples (e.g. containing BLOBs), the actual data is often stored in some other page (or even as a file) and only the pointer/file name is stored in the tuple.

Besides hash indices and B-trees, there are other indexing techniques like bitmaps, materialized aggregates (that allow checking whether the data is needed), or specialized indices (e.g. inverted indices).

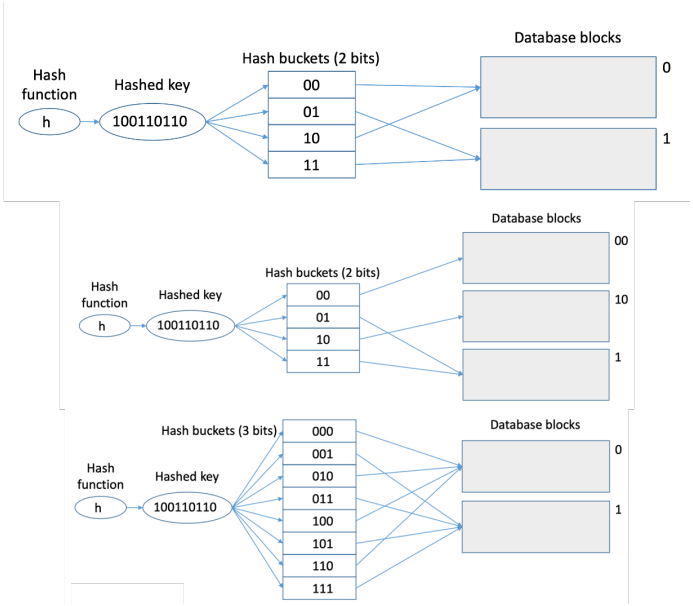
3.1 Hashing

Hashing is a very common operation in databases and used for internal data structures (buffer cache, lock table), operators (hash joins) and as an index/partition strategy. In a database, the hash function has to be computationally cheap and the hash table has to be big enough without wasting space (if it is too small, there are too many collisions). Mechanisms for handling collisions are:

- **Chaining**: The buckets of the hash table point to linked lists and a new entry is added to the lists on a collision.
- **Open Addressing**: On a collision, an empty slot is searched using some rule. For example linear probing where the next slot(s) are used or cuckoo hashing where several hash functions are used.

Growing/recreating the hash table is expensive (every item needs to be rehashed), this can be solved with:

- **Extensible Hashing**: Multiple hash buckets (i.e. hashed keys/parts of it) share the same database block. When a block gets full, it is split and only the items in the block need to be moved. The size of the table can be doubled by doubling the number of buckets (i.e. adding an additional bit to the hashed keys) and therefore increasing the degree of sharing.



- **Linear Hashing**: A split pointer is used to indicate which bucket will be split in case of an overflow. On an overflow, the block that overflows is chained, the bucket indicated by the pointer is split and the pointer is moved. Entries above the split pointer need to use a second hash function for the expanded range. Like this, the cost of reorganization is randomized: Eventually, the pointer will reach buckets with a chain and split them. The list of buckets grows page by page instead of doubling it (like in extensible hashing).

3.2 B-Trees

In a B-tree of order k, each node (except the root) has between $\frac{k}{2}$ and k child nodes. A B+-tree (used in databases) is a B-tree but data is at the leaves only. The leaf nodes of B-trees contain records with keys in disjoint key ranges, the root node and possibly intermediate nodes contain separator keys and pointers to the children. B-trees provide good perfor-

mance for exact-match queries, range queries, and full scans (with sorted output if desired). B-trees exploit disk pages by matching the node size to the page size (or even have nodes with sizes of multiple pages), therefore minimizing seek time/latency. Access to individual records requires a buffer pool.

The root contains at least one key and at least two child pointers, all other nodes are at least half full at all times (although this requirement can be violated in practical deployments where underflows persist and are resolved by defragmentations). In most designs that are used today, only the leaf nodes hold user data (which has been called B^+-tree but is nowadays the default design when B-trees are discussed). The records in leaf nodes contain a search key plus some associated information which can be all the columns associated with a table, a pointer to a record or anything else. Only child pointers are required, but many implementations also maintain neighbor pointers. B-tree nodes usually contain a fixed-format page header, a variable-size array of fixed-size slots, and a variable-size data area. The header contains a slot counter, information pertaining to compression and recovery, and more. The slots serve space management for variable-size records. Often more than 99% of all nodes are leaf nodes and the root and intermediate nodes are kept in memory.

A search requires one root-to-leaf pass where binary search is used within a single node. The overall number of comparisons is $\log_2(N)$, node size and record size do not influence it (asymptotically, only produce secondary rounding effects, but the record count is the only primary influence). A scan can employ neighbor pointers if they exist (but those scans are limited to one asynchronous prefetch at-a-time), otherwise parent/grandparent nodes are used.

If a leaf "overflows" after insertion, it is split and a new separator key is inserted into the parent (if the parent also overflows, this process continues potentially up to the root where a new level is inserted). Splits can be delayed by load balancing among siblings.

In databases, uniqueness constraints are often enforced with B-trees. If a B-tree structure (rather than a heap) is employed to store all columns in a table, it is called a primary index (or index-organized table). In secondary indices, entries contain a reference to a row/record in the primary index. The reference can be a search key (therefore requiring another search in the primary index) or a record identifier including a page identifier (which requires updates in secondary indices after splits in the primary index). Uniqueness of the keys can be achieved by uniqueness constraints or some artificial fields (e.g. timestamp for primary indices or the search key for

secondary indices). For non-unique values, the key can be repeated at the leaf nodes for every duplicated entry or the key can be stored once with a pointer to a linked list of all the matching entries.

Compared to hash indices, B-trees offer many advantages (space management, multi-field indices, range queries, ordered scans, better handling of skewed key value distributions, algorithms for efficient creation, phantom protection, etc...). And even with respect to I/O savings, they can only require one I/O (like hash indices) if the root and intermediate nodes are cached.

A clustered index forces the tuples to be stored in the same order as the index indicates (i.e. the table is physically stored in sorted manner), which is typically done only for the primary key and automatic in systems that store the data in the leaf nodes.

There is a tradeoff between tree depth (smaller nodes with less collisions) and breadth (larger nodes). For slow disks, larger nodes can be beneficial.

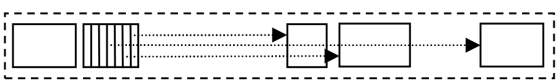
Because the access patterns for the index blocks often times looks different and to avoid that working on the index affects working on the data, there can be an own memory buffer for them.

3.2.1 Data Structures and Algorithms

The node size for B-trees should be optimized based on latency and bandwidth of the underlying storage and is therefore different for traditional storage (generally larger nodes) and flash devices (small nodes).

Instead of binary search, interpolation search can be used: Instead of inspecting the key in the center of the remaining interval (like binary search does), the position of the sought key value is estimated, typically using a linear interpolation based on the lowest/highest key value in the remaining interval. In the worst case, performance can be linear, therefore systems often (gradually) switch to binary search after a certain number of steps.

B-trees usually support variable-length records and variable-length separator keys. For records, an indirection vector (slot array) with entries of fixed size that contain the byte offset of the records is used. The indirection vector usually grows from left to right, the set of records from right to left (therefore enabling many small records or a few large ones):



The entries in the indirection vector are sorted on their search keys (to enable efficient binary search). Deletion typically leaves a gap in the data space, but must keep the indirection vector dense and sorted.

Keys are often transformed into a binary string such that binary comparisons suffice for sorting (normalized keys). But because they can be longer, they are sometimes only used in branch nodes and not in leaf nodes.

Prefix truncation stores the common prefix in a B-tree node only once (increasing the fan-out of branch nodes). The prefix is often not based on actual values, but on the maximal possible key range (which can be captured in the node with two additional fence keys). Dynamic prefix truncation refers to ignoring key bytes shared by the lower and upper bounds of the remaining search interval (e.g. in binary or interpolation search). In offset-value coding, the shared prefix is replaced by an indication of its length. Suffix truncation refers to choosing the shortest string for separators that separates the keys in the leaves (which may not be an actual key value in the leaves).

Cache faults for instructions can be reduced by use of normalized keys. Storing a few bytes of the normalized key (in B-tree leaves, e.g. the indirection vector) is called "poor man's normalized keys" and can speed up comparisons (if only the first bits are needed). The indirection vector can also be organized as a B-tree of cache lines ("B-trees within B-tree nodes") to minimize cache faults.

Although all keys are unique, duplicate values in the leading fields of a search key can be exploited to reduce storage space/search effort. Record lists (e.g. containing references) can be efficiently compressed using various schemes (prefix truncation, run-length encoding, bitmaps that are further compressed, etc...).

Other compression techniques are order-preserving Huffman codes or storing only the numeric difference of pointers.

For insertions (and similar for deletions), there is the choice between compaction (reclaiming free space within a page), compression, load balancing, and splitting (where all operations have different locality). SB-trees allocate disk space in large contiguous extents of many pages, leaving some free pages in each extent during index creation/defragmentation.

A split can cause a split in the parent, grandparent, ... up to the root which can conflict with searches of other threads. Lock coupling always locks a page and its parents to prevent problems when a page must be split. But this is not enough if splits are needed further up. A solution to that is to

check on every node if there is space for one more entry and split the node if not (which can be done because the parent was checked). Therefore, a split of a page never causes the changes to propagate back to the root. An alternative is to first try lock coupling and abort/start again from the root (locking the entire path) if a split beyond the parent is needed. In B^{link}-trees, splitting and posting a new separator key in the parent is divided into two steps. During the intermediate state, the new node is linked to the old neighbor, not its parent. As soon as convenient, this is cleaned up.

3.2.2 Transactions

There is the notion of "system transactions", transactions that modify, log, and commit changes in the database representation but have no effect on database contents (e.g. node splits). They do not need to be rolled back in order to ensure correct database contents.

Update locks are designed for applications that first test a predicate before updating a data record. An update lock permits only one transaction at at time in a state of indecision about its future actions. After predicate evaluation, either the update lock is upgraded to an exclusive lock or downgraded to a shared lock.

For deletions, specially marked ghost records (with an activated ghost bit) are usually used. Queries must ignore ghost records and locking absence (i.e. locking ghost records) is essential for serializability. If a ghost record is not locked, the space may be reclaimed. Adding very short ghost records to a page can also enable efficient insertions because less shifts are required in the indirection vector.

Key range locking locks key values and the gaps between key values which is required for serializability. Traditional designs lock the gap between two key values stored on neighboring B-tree leaves by accessing a neighbor node, even if that neighbor node cannot otherwise contribute to the query or update. This can be avoided with fence keys (copies of separator keys posted in parent nodes while splitting nodes) as the possible key range within a page is known with them. Key range locking can be applied to leaf pages or alternatively to separator keys (which can minimize the number of locks).

In a root-to-leaf traversal, there is a brief window of vulnerability between reading a pointer value and accessing the child node. This is solved by latch coupling where the latch on the parent page is retained until the latch on the child page is acquired.

In physical logging schemes, whole pages are logged. In logical logging, only insertion/deletion (including the content) without a reference to the physical location is logged. Physiological logging is "physical to a page, logical within a page". Pages are referenced by their physical address (page identifier) and records within pages are referenced by their slot number or their key value but not their byte address.

There are non-logged page operations (allocation-only logging/minimal logging) which means that the page content is not logged. However, the operations are reflected in the log (e.g. a split where the old page before the split can be employed to ensure recoverability of both pages).

Most systems have facilities for non-logged creation of secondary indices where the content of B-tree pages is not logged (only changes in the catalogs/free space management information). In these cases, the new index is forced to storage upon completion.

For online index creation, there are two principle designs. The concurrent updates can be applied to the structure still being built ("no side file") or captured elsewhere and applied after the main index creation activity is complete ("side file"). In the "no side file" design, deletions are represented by "anti-matter" records. In "side file" designs, "catch-up" operations based on a log of the updates are needed.

3.2.3 Query Processing

B-tree indices are exploited in query execution plans for retrieval and ordered scans ("index-order scan"). A "disk-order scan" is guided by allocation information for the B-tree (e.g. kept in bitmaps) and may be faster for reading all tuples if sorted output is not needed. If the B-tree is heavily fragmented, it may even be faster if fewer than all leaves are required. Coordinated scans can speed up scans. If another scan is already active, the scans are linked and the new scan starts at the current position of the prior scan. Furthermore, pages that remain in the buffer pool can be exploited and overall system throughput can be maximized by sorting required future I/O by relevance.

Secondary indices are valuable for selective queries, but can result in a large number random I/O operations for larger result sets. Scanning the table costs practically the same, independent of the result size. A plan with robust performance (i.e. a scan) may be preferable unless cardinality estimation is very accurate.

If a secondary index can supply all the columns required in a query, or all columns from a particular table, then there is no need to fetch records

from the heap or the primary index. The common terms for this are "index-only retrieval" or "covering indices". Some systems permit adding columns (e.g. primary key/foreign key columns or dates in business intelligence databases) to an index definition that are not part of the search key to permit index-only retrieval in more queries. If multiple secondary indices together cover a given query, they can be joined on the common reference column.

If an index needs to be probed with many search keys, sorting the keys (i.e. the intermediate result) avoids duplicate searches and may improve buffer pool effectiveness and thus I/O costs. But if the secondary index is used to provide an ordering that is used in another operator (e.g. merge join), the fetched records must be sorted back into the order in which they were obtained by the secondary index. As another improvement, an index search may resume where the preceding search left off.

Multi-column indices support equality predicates on any prefix of their columns. But even for subsets of the columns, some optimizations are possible, e.g. exhaustive enumeration of the missing column or transformations of the query predicate.

In addition to any prefix, a B-tree can produce sorted output on many other variations of its column list using merging (if a sort is requested on B, C but only A, B, C is indexed, the values of A can be enumerated and the runs for different values of A merged) or segmented execution (if a sort is requested on A, B, P, Q but only A, B is indexed, an individual sort operation can be performed per value combination of A, B).

Secondary indices can be used for semi-join reduction. Secondary indices from different tables (that cover the join predicates) are joined first based on the join predicate which reduces the list of rows to be fetched (not only by single-table selection, but also by multi-table predicates). This is especially useful in a star schema because the number of fetched rows from the fact table can be reduced significantly depending on the predicate (that is based on the dimension tables).

A join index maps values of join columns to references in two (or more) tables and therefore can speed up joins. A "star index" is similar but contains record references to the dimension tables rather than key values.

For updates, there is the row-by-row and the index-by-index technique. In the row-by-row approach, all indices are updated before the next row is processed. The index-by-index strategy applies all changes to the primary index first (generally sorted, which makes this approach especially useful if there are more changes than pages as each index page is written at

most once). Since a sort operation creates separate plan phases, this can also protect from the Halloween problem.

Tables and indices can be partitioned horizontally (by rows) or vertically (by column, i.e. columnar storage). Partitions can be kept in its own data structure (partitioned table/index) or in a single B-tree (partitioned B-tree). A secondary index is local if it is partitioned horizontally in the same way as the primary data structure of the table, in which case references do not require partition identifiers.

3.2.4 Utilities

The performance of index creation is much improved by first sorting the future index entries (and creating it from the bottom up, i.e. creating the inner nodes based on the bottom nodes min/max values up to the root). Commands for index creation usually have many options (e.g. compression, leaving free space for future updates, temporary space for sorting future entries, etc...).

Index removal can be complex, especially if some structures must be created in response. It can be instantaneous by marking them as obsolete and delaying the actual deletion/deallocation.

Indices may be rebuild after a corruption or when defragmentation is desired but removing and rebuilding is faster. Ignoring the rules of the B+ tree and not merging nodes / delaying merges (and periodically rebuilding the tree) can result in better performance. When a primary index is rebuilt, the secondary indices usually must be rebuilt too (because all references change).

Bulk insertions can be sped up by buffering the insertions in the root node or in branch nodes. Another optimization technique are partitioned B-trees where recent insertions are sorted in-memory and appended as a new partition. Because efficiency of index maintenance is so poor in some implementations, some vendors recommend dropping indices before bulk insertions.

Partitioned B-trees can also help with bulk deletions by moving records to a dedicated "victim" partition which is deleted very efficiently. To efficiently index data streams, techniques for bulk insertions and bulk deletions are needed.

Most implementations of B-trees require occasional defragmentation (reorganization) to ensure contiguity (for fewer seeks during scans), free space, etc. Defragmentation includes for each node free space consolidation within each page, removal of ghost records, and optimization of

in-page data compression. Not using neighbor pointers (but fence keys instead) and careful write-ordering can reduce the cost of page movement.

Complete, reliable, and efficient verification of B-tree structures is a required defensive measure and can be done with an index-order sweep. If this is not desirable, the B-tree pages can be scanned in any order, required information can be extracted and matched with information in other pages (i.e. algorithms based on aggregation of facts extracted from pages). As a side effect, query execution may verify all B-tree invariants.

3.2.5 Advanced Key Structures

By careful and creative construction of B-tree keys, additional indexing capabilities can be enabled.

If the attribute being indexed is a sequence (e.g. time series), the tree only grows from the right side and all values go to the same block. In a reverse index, the key is reversed before inserting and sequential values therefore go to different pages. A disadvantage is that range queries on the attribute are no longer efficient.

Multidimensional UB-trees are B-tree indices on space-filling curves, each point and range query against the original dimensions is mapped to the appropriate intervals along the curve.

Partitioned B-trees are achieved with an artificial leading key. Usually, the same single value appears in all records and there are techniques that exploit alternative values, but only temporarily. Those trees are useful for efficient sorting, index creation, bulk insertion, and bulk deletion.

Merged indices are B-trees that contain multiple traditional indices and interleave their records based on a common sort order. They therefore shift de-normalization from the logical level of tables and rows to the phyisical level of indices and records. By clever design of the keys, the sort oder alone keeps related records co-located without additional pointers between records, for instance in the following record sequence of a merged index:

…
Order 4711, Customer “Smith”, …
Order 4711, Line 1, Quantity 3, …
Order 4711, Line 2, Quantity 1, …
Order 4711, Line 3, Quantity 9, …
Order 4712, Customer “Jones”, …
Order 4712, Line 1, Quantity 1, …
…

Columnar storage may also be based on B-trees. A tag value (row identifier) is used when storing the columns as the key. Because these tags are usually consecutive (logical) identifiers, only a single tag value is required per page and the other values can be calculated by the offsets.

B-trees can also be used to store large objects (allowing efficient insertions and deletions of byte ranges) where the byte offsets are used as key values in the intermediate nodes.

Appending a version number (which can be a time delta) to each key value and compressing neighboring records as much as possible (e.g. storing only the difference between a version and its predecessor) turns B-trees into a version store.

3.3 Column-Oriented Database Systems

In column-oriented database systems/column-stores, each attribute of a table is stored in a separate file or region on storage. A column-store can therefore access just the columns needed for a query, which can be especially useful for analytic queries that perform scans and aggregates over a few columns. However, for queries that only access a single record, a column-store will have to seek several times (whereas row-stores require only one seek). They are also not ideal when most of the attributes are needed anyway, when the tuple or parts of it needs to be reconstructed and for updates/modifications.

A typical implementation for storing records inside a page (in a row-oriented system) are slotted-pages which is also known as the N-ary storage model (NSM). The decomposition storage model (DSM), where each column is stored separately and for each value the virtual ID is stored explicitly, is a historical predecessor of column-stores. The first major column-store was MonetDB with the original motivation to address the increasing (compared to the CPU) latency of memory requests. PAX adopted a hybrid NSM/DSM approach and fractured mirrors stored one copy of the data in NSM format and a separate copy in DSM. Similarly today, SAP HANA stores data both in a row-format and in a column-format.

In C-Store, data on disk is represented as a set of (compressed) column files which is known as the "read optimized store" (ROS). Newly loaded data is stored (uncompressed) in a write-optimized store (WOS) that is periodically flushed to the ROS and queries need to access the ROS and WOS. The system supports multiple projections and efficient indexing into sorted projections using sparse indices that store the first value contained on each physical page of a column.

MonetDB relies solely on memory mapped files and avoids the overhead/-complexity of a buffer pool. Its execution engine uses a column-at-a-time algebra (BAT algebra) and the query plans therefore provide the CPU more in-flight instructions, keep the pipelines full and the branch misprediction/CPU cache miss rates low. BAT stands for binary association table and refers to a two-column (surrogate, value) table where the surrogate is a non-materialized virtual ID. Intermediate and the final results are always stored in BATs, the BAT algebra operators consume and produce BATs and perform a fixed hard-coded action on a simple array. The philosophy can also be paraphrased as "the RISC approach to database query languages": With a simple algebra, opportunities are created for implementations that execute the common case very fast. For updates, a collection of pending updates columns for each column is used.

VectorWise improves MonetDB by introducing compression and a vectorized execution model (where one block/vector of a column is processed at a time) that strikes a balance between full materialization of intermediate results (like in MonetDB) and the high functional overhead of tuple-at-a-time iterators in traditional systems. Positional delta trees are used for updates (see below).

Besides these three major implementations, it is also possible to use columnar storage only but immediately stitch data into tuples and fed them to classic row-store engines. Such a design is easy to implement, but cannot exploit many benefits of column-stores.

In the following subsections, various optimization techniques for column-stores are summarized. Many of those are not specific to column-stores and can be applied in one way or another also to row-store systems. However, because column-stores are designed substantially different from traditional row-stores, they are able to maximize the benefits of these ideas.

3.3.1 Virtual IDs

The simplest way to represent a column in a column-store is to assign a tuple identifier with every column. This ID is often not materialized and the position/offset of the tuple is used instead.

3.3.2 Block-oriented/Vectorized Processing

Column-Stores can achieve better cache utilization by passing cache-line sized blocks of tuples between operators. Furthermore, vectorized CPU instructions can be used to operate on multiple values at a time. Regarding control flow, the operators are similar to those in tuple pipelining (but

the `next()` method returns a vector of tuples). Regarding data processing, the primitive functions look much like MonetDB's BAT algebra. The size of vectors is typically chosen such that they fit in L1 cache. Vectorized processing has many advantages:

- Reduced interpretation overhead (less function calls)
- Better cache locality
- Compiler optimization opportunities (tight loop over arrays, SIMD)
- Optimized algorithms that perform better on whole blocks
- Parallel memory access
- Less profiling overhead
- Adaptive execution (because of branches, it can be faster to compute a result for the whole vector, even if some elements are not needed)

Vectorized execution can also be applied to row stores/mixed formats and the tuple layout can be changed during execution (which means that optimizers can determine the best layout with query layout planning).

3.3.3 Late Materialization

The joining of columns into wider tuples can be delayed or avoided completely for some queries. In these cases, column-stores not only store data one column-at-a-time, but also process data in a columnar format.

Intermediate "position" lists (that are e.g. intersected and used for extraction in another column) often need to be constructed in order to match up operations that have been performed on different columns. Enforcement of sequential access patterns and tuple alignment across columns reduces the costs of tuple reconstruction.

Late materialization can improve memory bandwidth efficiency/cache performance and allows operating directly on compressed data. But it can sometimes be slower than early materialization, e.g. with non-restrictive predicates where large amounts of positional intermediate data needs to be intersected. Multi-column blocks store horizontal partitions of data in groups of columns (that are called multi-column blocks, vector blocks, or column-groups). The position list results from the predicate application are pipelined to an intersection operator and the resulting bitmap is stored in the data structure itself (position descriptor). This can allow the construction of tuples while the corresponding values are still in cache.

3.3.4 Column-specific Compression

Each column can be compressed using a method that is most suitable for it and values from the same column tend to have more value locality than values from different columns. As CPU performance increases much faster than memory bandwidth, compression becomes even more useful. It can also transform variable-sized data into fixed-width arrays which is beneficial for SIMD instructions.

Frequency partitioning reorganizes a column such that the information entropy in each page is as low as possible, therefore increasing the compression efficiency.

Many compression algorithms can be used. Run-length encoding (RLE) compresses runs of the same value in a column to a compact singular representation of (value, start position, run length). When columns have a limited number of possible data values, bit-vector encoding can be useful which creates a bitmask for every possible value of the field (which indicates if position i has this value). Dictionary encoding works well for distributions with a few very frequent values. A per-block or global dictionary can be used and this encoding potentially allows rewriting of predicates on strings into faster predicates on integers. Furthermore, it can result in fixed width columns. When value locality is high, frame of reference (FOR) can be used where values are represented with some constant base plus a value. This can be combined with delta encoding for even better compression.

3.3.5 Direct Operation on Compressed Data

Working over compressed data can improve utilization of memory bandwidth. For instance, a sum over a run-length encoded integer can be replaced with a product. Or if order preserving encoding is used, comparison actions can work with the encoded value. To facilitate implementation/extension, compression blocks that provide a unified API to query operators (e.g. `getSize()`, `isSorted()`, etc...) can be used.

3.3.6 Efficient Join Implementations

With late materialization, more efficient join implementations are possible in some cases. For instance, only the columns that compose the join predicate can be the input of a join. The output of the join is a set of pairs of positions in the two input relations. At least one set of output positions will be unsorted, which is problematic for further operations. A solution to that is a "jive join", which sorts the positional list while remembering the

original order (that must be preserved to match up with the join output from the other table). The algorithm requires two sorts but to reduce the random access performance overhead, a complete sort is not needed (as the data is accessed in blocks, so the list only needs to be partitioned into the blocks on storage). Radix join exploits this fact.

An alternative is to use a hybrid materialization approach where the right (inner) table is materialized early but the left (outer) table, where the positional list is (usually) sorted, is not. The right (inner) table can also be represented using multi-column blocks.

3.3.7 Redundant Representation of Individual Columns in Different Orders

Several copies of each column can be stored sorted by attributes heavily used in an application's workload. Groups of columns sorted on a particular attribute are called projections, virtual IDs are on a per-projection basis.

3.3.8 Database Cracking/Adaptive Indexing

Projections are a form of indexing and the storage overhead is typically not that large because columns compress very well. However, there is an update overhead because all projections need to be updated. Another form of indexing are zonemaps that store light-weight metadata per page like min/max information.

With traditional indices, fixed up-front decisions need to be made regarding which indices should be created and the creation can take a long time. With adaptive indices, they are created adaptively, partially, and continously. One such type of index is database cracking. The physical data store is continously changing with each incoming query q, using q as a hint on how data should be stored. For instance when a query with predicate $A < 10$ comes in, all tuples of A with $A < 10$ are clustered at the beginning of the column. This partitioning can then be further refined with consecutive queries. With cracking, a database can immediately start processing queries without any preparation while reaching optimal performance after a few queries.

3.3.9 Efficient Loading Architectures

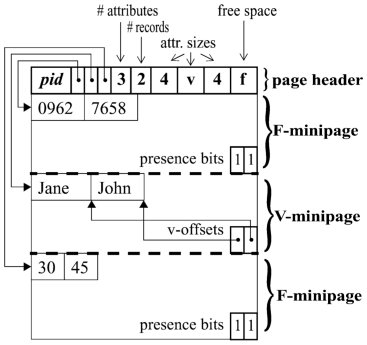
Column-stores are more sensitive to updates because each record/tuple is stored in more than one file. Besides the ROS/WOS approach in C-Store and the collection of pending updates in MonetDB (where parts of a query

can be performed on the read-store/delta data separately), positional delta trees like in VectorWise can be used for updates/insertions. In this case, an update query immediately finds out which table positions are affected (and stores this information efficiently). The activity of merging is therefore moved from query time to update time.

3.4 PAX

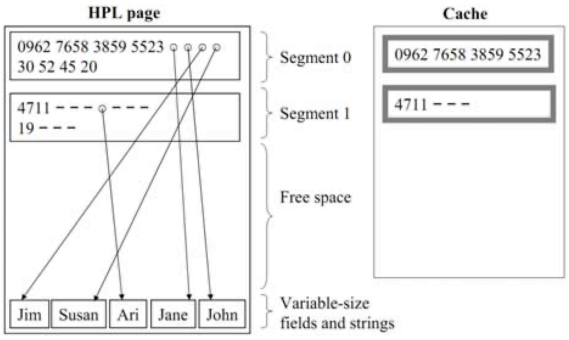
Transferring data from the memory to the CPU caches can incur significant delays. Because many query operators access only a small fraction of each record, NSM can pollute the cache with useless data, therefore wasting bandwidth and possibly forcing replacement of other information. The cache behavior of DSM is generally better, but it can require expensive record-reconstructions, especially when many columns are requested. Patrition Attributes Across (PAX) tries to repair NSM's cache behavior, without compromising its advantage over DSM. It stores the same data on each page as NSM, but groups all the values of a particular attribute together on a minipage within each page. To reconstruct records, a mini-join with minimal costs among minipages is needed. Implementing PAX on a DBMS with NSM requires only changes on the page-level data manipulation code.

At the beginning of each page there is a page header that contains offsets to the beginning of each minipage. Fixed length attribute values are stored in F-minipages with a presence bit vector at the end of each F-minipage (denoting null values). Variable-length attribute values are stored in slotted V-minipages with pointers to the end of each value (and null pointers for null values). PAX requires the same amount of space as NSM to store a relation. Upon record deletion, PAX reorganizes minipage contents to fill the gaps in order to minimize fragmentation which could hurt cache utilization.



3.4.1 HPL

One disadvantage of the PAX format is the complexity of free space management within each page for records with one or more variable-size fields. In-page optimizations may be required after the page is filled when one or more of the minipages run out of space. The hybrid page layout (HPL) tries to preserve the simplicity of two variable-size allocation spaces growing towards each other (from NSM) and the cache efficiency during large scans (from PAX). It does so by introducing an allocation space for fixed-size fields and an allocation space for variable-size fields that grow from both ends towards each other. Offsets for variable-size fields are considered fixed-size fields. For fixed-size fields, segments are introduced where values of the same field are grouped in one cache line. Therefore, as in PAX each cache line contains values from only one field but the cache lines holding the same field are distributed over all segments within a page. Because fetching multiple contiguous cache lines is generally more performant, it is desirable to have a small number of cache lines per field in a segment. But this (and bitmaps for ghost records/null values) can cause the segments to be quite large. Incremental segments reduce the size of segments by not using all available space in cache lines for small values, therefore keeping the record count per segment small if desired.



Chapter 4

Query Processing

Given a query, there are many access plans that a DBMS can follow to process it and produce its answer. All plans are equivalent in terms of their final output but vary in their cost, that is, the amount of time that they need to run. Query processing can be roughly divided into:

- Query Parsing (including validation, access control, cache checks, etc...)
- Rewriting
- Optimization
- Code Generation
- Query Execution (interpreted or compiled)

In query processing, the resources must be shared and there is often a tradeoff between overall throughput and response time of an individual query.

Caching is an important optimization technique. Parsing/optimization (shared SQL area in Oracle DB), intermediate results (server result cache in Oracle DB), and query results outside the engine (e.g. in a key-value store; reduces the number of connections to the database) can be reused. Clients can also cache the results, sometimes with support from the server (that actively keeps the cache consistent). Consistency/stale results can become an issue when caching is used and one needs to decide on pre-populating caches or filling them on-demand.

For memory, we differentiate between memory local to a server process/-worker (program global area, with parts for query processing, session information, parameters, cursor status, etc...) and shared memory (system global area for shared resources such as the buffer cache, redo log buffer,

and large memory pool). Cursors are pointers to tables, indicating the next tuple to be delivered. They are declared over tables, views, results, etc...

4.1 Execution Models

The execution is modelled as a tree of operators, with the tables at the leaves and the query results at the root. Typically, each operator has at most two inputs from lower layers. Tables and queries are isomorphic, meaning that any subtree of the query can be materialized as a table and queries can create tables that are used as inputs for other queries (views). Operators in the tree function independently; they read input, process it, and produce an output. Some operators might be blocking (e.g. sorting), others can issue result tuples in a pipeline fashion. The three basic single machine execution models are:

- **Iterator Model (Volcano/pipeline execution model)**: Tuples traverse the tree from leaves to the root and operators iterate over those tuples to process them. An operator returns `NULL` when it cannot produce more tuples. The data therefore flows bottom up, the control top down. The model is generic and does not favor any workload, it provides a generic interface for all operators (which is easy to implement), there are no overheads in terms of main memory, and pipelining/parallelism is supported. However, there is a high overhead of method calls and poor instruction cache locality.
- **Materialization Model**: Operators produce all of their output in one go and make it available in some buffer which is used by the next operator. This works well in OLTP with small queries/transactions that do not process a lot of data. It is potentially more efficient for short queries, but not suitable for analytics with large data that is passed from operator to operator.
- **Vectorization/Batch Model**: Operators work on sets of tuples rather than single tuples of the whole input, they read sets of input tuples and produce sets of result tuples. This can exploit SIMD/AVX and columnar storage and is a combination of the iterator/materialization model. The number of calls are reduced compared to the iterator model and certain operators are more efficient (because of SIMD). The model works best for analytical queries and is widely used in data warehouses (OLAP).

We distingiush between pull mode and push mode. In pull mode, data is obtained by one operator through a function call to a lower operator. In

push mode, tuples/buffers/vectors are sent up and received from operators higher in the tree (which have to buffer them if they are not ready for processing). This can be more efficient, but is more difficult to implement (similar to event based programming), buffering capabilities are needed at all operators, and more synchronization is required.

For parallel processing, the models can be extended with an `EXCHANGE` operator that moves data from one place to another. This can be implemented as a "driver" on top of an operator that calls `next` several times, collects the results, sends the results to the other side, and the other side uses the data to respond to `next` calls to the original operator. All of this is transparent to the operator.

4.2 Views/Schemas

Schemas impose a fixed view on the data, views allow different views on it. A view results in a virtual table being added to the schema, materialized views result in actual table being added to the schema (where the data of the table is derived from base tables). Views are used to implement logical data independence, make it easier to write applications/queries (with views as intermediate stages), for access control, and to speed up processing with materialized views.

Databases for analytics (e.g. the TPC-H decision benchmark) often use a star schema. There is a (big) fact table with the central element of the schema and dimension tables (typically smaller) with more data on the different attributes that are mentioned in the fact table. In a snowflake schema, the dimension tables are normalized. Modern analytics (e.g. the TPC-DS benchmark) use a snow-storm schema with multiple fact tables and joins between them (fact to fact joins).

Queries are usually generated by applications (with templates), so they might not be the most efficient way to express the information need or be even logically impossible. For this reason and because of views, query rewriting can be important. Optimizing views on their own and embedding their plan into the global plan is unlikely to be optimal (e.g. because the optimal join order might differ for queries).

4.3 Query Rewriting

Query rewriting refers to transforming the original SQL query into an equivalent query that is more efficient (by removing operations), gives the query optimizer more freedom to operate, makes more explicit what the

query wants to do, and/or maps the query to actual base tables/views for efficiency reasons. It is typically done just looking at the schema (without cost estimates/statistics). Some important rewriting rules are:

- **Predicate Transformation**: If the number of needed comparisons can be reduced, the query will run faster. Two possible transformations are:

```
SELECT * FROM T WHERE (T.price > 50 AND T.price < 100) OR (T.price >
    90 AND T.price < 200)
SELECT * FROM T WHERE (T.price > 50 AND T.price < 200)
```

```
SELECT * FROM employee WHERE deptno = 'D11' or deptno = 'D21' or
    deptno = 'E21'
SELECT * FROM employee WHERE deptno in ('D11', 'D21', 'E21')
```

  In the second example, the second option is preferred if there is no index over `deptno`.

- **Predicate Augmentation**: The rewriter can augment the query with the "transitive closure" by using logical facts like $A = B \wedge A > c \Rightarrow B > c$, e.g.:

```
SELECT empno, lastname, firstname, deptno, deptname FROM employee emp,
     department dept WHERE emp.workdept = dept.deptno AND dept.deptno
    > 'E00'
SELECT empno, lastname, firstname, deptno, deptname FROM employee emp,
     department dept WHERE emp.workdept = dept.deptno AND dept.deptno
    > 'E00' AND emp.workdept > 'E00'
```

```
SELECT * FROM T, R, S WHERE T.id = R.id AND R.id = S.id
SELECT * FROM T, R, S WHERE T.id = R.id AND R.id = S.id AND T.id = S.
    id
```

- **Arithmetic**: Arithmetic operations can be rewritten, e.g. deriving an average from a sum and the count (if those are calculated anyway), therefore reducing the number of calculations.

- **Predicate Pushdown/View Folding**: Queries involving views can be rewritten such that they work on the base table with the appropriate predicates applied to the base tables (instead of the views). This prevents going over the data twice and is the main reason that views are cheap. If the resulting query involves a join on the same table (because the query joined two views on the same base table), the join can be eliminated.

- **Unnesting of Queries**: Checking agains nested queries can be expensive, unnesting (which e.g. results in a join) can therefore be useful:

```
SELECT empno, firstnme, lastname, phoneno FROM employee WHERE workdept
     in (SELECT deptno FROM department WHERE deptname = 'OPERATIONS')
SELECT empno, firstnme, lastname, phoneno FROM employee emp,
    department dept WHERE emp.workdept = dept.deptno and dept.deptname
     = 'OPERATIONS'
```

- **Optimization using Views**: When a view is materialized, it can be useful to replace the base table with the view (even for partial matches).

4.3.1 Relational Algebra

Query rewriting (and transformations done by the optimizer) are possible because there exists a formalism to prove equivalence among queries. The role of the optimizer is to consider as many as possible of these equivalent queries (plans) and query rewriting helps by simplifying the way queries are represented (e.g. view folding), giving the optimizer more options (e.g. predicate augmentation), and removing syntactic sugar/programming artifacts (e.g. unnesting of queries). Some equivalence rules for relational algebra are:

1. **Deconstruction of conjunctive selection operations**:

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. **Commutative selection operations**:

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_2 \wedge \theta_1}(E)$$

   It can therefore be beneficial to apply the most selective predicate first.

3. **Combination of selections with cartesian products and theta joins**:

$$\sigma_{\theta}(\mathrm{E}_1 \times \mathrm{E}_2) = \mathrm{E}_1 \bowtie_{\theta} \mathrm{E}_2$$

$$\sigma_{\theta_1}(\mathrm{E}_1 \bowtie_{\theta_2} \mathrm{E}_2) = \mathrm{E}_1 \bowtie_{\theta_1 \wedge \theta_2} \mathrm{E}_2$$

4. **Commutativity/Associativity of joins**:

$$E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$$

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

If θ_2 involves attributes from only E_2 and E_3:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

5. **Distribution of selection operation over the theta join**: When all the attributes involve only the attributes of one of the expressions (E_1):

   $$\sigma_{\theta_0}(\mathrm{E}_1 \bowtie_\theta \mathrm{E}_2) = (\sigma_{\theta_0}(\mathrm{E}_1)) \bowtie_\theta \mathrm{E}_2$$

   When θ_1 only involves the attributes of E_1 and θ_2 only those of E_2:

   $$\sigma_{\theta_1} \wedge_{\theta_2} (\mathrm{E}_1 \bowtie_\theta \mathrm{E}_2) = (\sigma_{\theta_1}(\mathrm{E}_1)) \bowtie_\theta (\sigma_{\theta_2}(\mathrm{E}_2))$$

   This enables predicate pushdown.

6. **Distribution of projection operation over the theta join**: If θ only involves attributes from $L_1 \cup L_2$:

   $$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

   This enables projection push down.

7. **Set operations**: Union/Intersection are commutative and associative, the selection operation distributes over $\cup$, $\cap$, and $-$:

   $$\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2)$$

   For $\cap$ and $-$ also:

   $$\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - E_2$$

   Furthermore, the projection operation distributes over the union operation:

   $$\Pi_{\mathrm{L}}(E_1 \cup E_2) = (\Pi_{\mathrm{L}}(E_1)) \cup (\Pi_{\mathrm{L}}(E_2))$$

4.4 Query Optimization

Query optimizers use rules/heuristics and cost models to decide which plans are the best. There are many choices to be made when coming up with the best plan, e.g. the access method (depending on indices, predicates, clustered tables), operator implementation (depending on sorted input data or results), or the shape/form of the query tree (pushdown of selections/projections, join orders). The algebraic space determines the orderings of the necessary operators to be considered whereas the method-structure space is about the implementation choices for the execution of each operator. The estimator uses statistics from the dictionary to compute the costs. The statistics are constantly collected, typical ones are:

- **Table statistics** (number of rows and blocks, average row length)
- **Column statistics** (number of distinct values and nulls, data distribution informaton/histograms)
- **Extended statistics** (index statistics, number of leaf blocks, levels, clustering factor; i.e. how ordered the data is)
- **System statistics** (I/O and CPU performance/utilization)

In a join, we generally want to have the smaller table as outer table such that the access to the larger table is sequential. In a hash join, the smaller table is used to build the hash table and the big table for probing.

All major databases use histograms for cardinality estimation. We differentiate between equi-width histograms where the ranges of values are fixed and equal (which helps identifying hot-spots) and equi-depth/equi-height histograms where the number of tuples per bucket are equal (which helps to partition the data evenly). A singleton/frequency histogram plots the frequency of every distinct item in a table which is very useful to compute the selectivity of a query, but can only be done if the number of distinct values is not too high. Zone maps (e.g., in Snowflake) are a combination of a coarse index and statistics that keep statistics per block and can sometimes replace indices.

The notion of cardinality is used in different contexts in databases:

- **Attribute Cardinality**: Number of distinct values for the attribute (estimated using statistics)
- **Table Cardinality**: Number of tuples in the table
- **Operator Cardinality**: Number of tuples that must be processed to get the result (e.g. $n \times m$ for a nested loop join, estimated using the input sizes and operator types/access methods)
- **Predicate Cardinality**: Number of tuples that match the predicate

Cardinality estimation for intermediate results can be very hard.

The selectivity of an operator is how much data it will produce (where a selectivity value of 1 means that all the data will be selected and 0 none). Note that when we talk about a "high selectivity", a lower selectivity value is meant, i.e. that not much tuples are selected. Calculating selectivity for keys is very easy (distinct values), whereas we can assume uniform data distribution for equality and range predicate (and therefore calculate the estimated selectivity) or use a histogram for the estimate. For negation predicates, we subtract the selectivity of the non-negated predicate

from 1. When we have multiple predicates, independence can be assumed (where the selectivities are multiplied for conjunctions and we add the selectivities and subtract the selectivity of the intersection for disjunctions). For correlated attributes, the estimation is more difficult (correlation can be guessed or combined histograms can be used).

4.4.1 Rule-Based Optimization

Rule based optimizers do not look at statistics or the contents of the tables, they only use transformation rules, (static) ranks of what methods are preferred for doing an operation, and schema information. The rules are based on experience, but a rule based optimizer can make mistakes because it ignores the actual data. For the access methods, there are many possible choices:

- Full table scan
- Sample table scan (read a sample of the table)
- Table access by row ID (very fast, happens often for OLTP queries)
- Cluster scans (reading the blocks where the tuples with the same value for the cluster key are stored; assuming rows are grouped together based on a cluster key)
- Hash scans
- Index scans (can be bounded/unbounded with different ranks)

A rule-based optimizer will choose the available (determined based on predicates) access plan with the highest rank for a given query.

For joins, different join orders are generated and one is chosen according to a criterion. In Oracle DB, the goal of the optimizer is to maximize the number of nested loop joins in which the inner table is accessed using an index scan, which is achieved by various heuristics (for generating the considered orders and choosing among them).

4.4.2 Cost-Based Optimization

The basic approach is to consider all possible ways to access the base tables and to execute the query (join orderings), evaluate the cost of every possible plan (according to a cost model), and pick the best plan. However, the space to explore can be huge, query optimization is NP-hard, costs are only approximate and the time to optimize a query is limited. The most expensive part is plan enumeration (mainly determining the join order) and there are multiple approaches like dynamic programming (often

- **Table statistics** (number of rows and blocks, average row length)
- **Column statistics** (number of distinct values and nulls, data distribution informaton/histograms)
- **Extended statistics** (index statistics, number of leaf blocks, levels, clustering factor; i.e. how ordered the data is)
- **System statistics** (I/O and CPU performance/utilization)

In a join, we generally want to have the smaller table as outer table such that the access to the larger table is sequential. In a hash join, the smaller table is used to build the hash table and the big table for probing.

All major databases use histograms for cardinality estimation. We differentiate between equi-width histograms where the ranges of values are fixed and equal (which helps identifying hot-spots) and equi-depth/equi-height histograms where the number of tuples per bucket are equal (which helps to partition the data evenly). A singleton/frequency histogram plots the frequency of every distinct item in a table which is very useful to compute the selectivity of a query, but can only be done if the number of distinct values is not too high. Zone maps (e.g., in Snowflake) are a combination of a coarse index and statistics that keep statistics per block and can sometimes replace indices.

The notion of cardinality is used in different contexts in databases:

- **Attribute Cardinality**: Number of distinct values for the attribute (estimated using statistics)
- **Table Cardinality**: Number of tuples in the table
- **Operator Cardinality**: Number of tuples that must be processed to get the result (e.g. $n \times m$ for a nested loop join, estimated using the input sizes and operator types/access methods)
- **Predicate Cardinality**: Number of tuples that match the predicate

Cardinality estimation for intermediate results can be very hard.

The selectivity of an operator is how much data it will produce (where a selectivity value of 1 means that all the data will be selected and 0 none). Note that when we talk about a "high selectivity", a lower selectivity value is meant, i.e. that not much tuples are selected. Calculating selectivity for keys is very easy (distinct values), whereas we can assume uniform data distribution for equality and range predicate (and therefore calculate the estimated selectivity) or use a histogram for the estimate. For negation predicates, we subtract the selectivity of the non-negated predicate

from 1. When we have multiple predicates, independence can be assumed (where the selectivities are multiplied for conjunctions and we add the selectivities and subtract the selectivity of the intersection for disjunctions). For correlated attributes, the estimation is more difficult (correlation can be guessed or combined histograms can be used).

4.4.1 Rule-Based Optimization

Rule based optimizers do not look at statistics or the contents of the tables, they only use transformation rules, (static) ranks of what methods are preferred for doing an operation, and schema information. The rules are based on experience, but a rule based optimizer can make mistakes because it ignores the actual data. For the access methods, there are many possible choices:

- Full table scan
- Sample table scan (read a sample of the table)
- Table access by row ID (very fast, happens often for OLTP queries)
- Cluster scans (reading the blocks where the tuples with the same value for the cluster key are stored; assuming rows are grouped together based on a cluster key)
- Hash scans
- Index scans (can be bounded/unbounded with different ranks)

A rule-based optimizer will choose the available (determined based on predicates) access plan with the highest rank for a given query.

For joins, different join orders are generated and one is chosen according to a criterion. In Oracle DB, the goal of the optimizer is to maximize the number of nested loop joins in which the inner table is accessed using an index scan, which is achieved by various heuristics (for generating the considered orders and choosing among them).

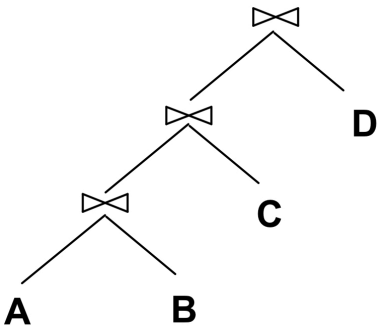
4.4.2 Cost-Based Optimization

The basic approach is to consider all possible ways to access the base tables and to execute the query (join orderings), evaluate the cost of every possible plan (according to a cost model), and pick the best plan. However, the space to explore can be huge, query optimization is NP-hard, costs are only approximate and the time to optimize a query is limited. The most expensive part is plan enumeration (mainly determining the join order) and there are multiple approaches like dynamic programming (often

used in practice), greedy heuristics, randomized algorithms, restricting the search space (only considering particular structures), or other heuristics like programmer hints.

The cost of an access method is query-, data-, and system-dependent. For a query with a high selectivity, an index scan can have lower costs whereas a full table scan can be the best access method if the whole table needs to be read. The optimizer usually keeps the best access methods according to the cost and the order. The order is considered because of interesting sort order, a particular sort oder of tuples that could be useful for a later operation (for instance for a merge-join). Only orders that are relevant to the query are considered, if tuples are ordered by an attribute where the sorting is useless (because the attribute does not appear in an operator that can benefit from orders), the optimizer treats this as non-sorted.

System-R only considers left-deep join trees which allow pipelining (no writing of intermediate results to temporary files):



Enumeration is done using N passes (for N relations). First, the best 1-relation plan (i.e. access method) for each relation is determined. Then, the best way to join the result of each 1-relation plan (as outer relation) to another relation is evaluated, i.e. the best 2-relation plans are considered. This is continued up to the N-relation plans. For each subset of relations, the overall cheapest and cheapest plan for each interesting order is kept. ORDER BY, GROUP BY, aggregates, etc... are handled as a final step, using a plan with an interesting order or with an additional sorting operator and cartesian products are avoided if possible.

Oracles optimizer first determines whether joining two or more of the tables results in a row source with at most one row. Furthermore, for joins with outer join conditions, the table with the outer join operator must come after the other table of the join (pushing the outer join as much up

as possible because it does not filter out any tuples). The optimizer also considers factors like the sort area size (increasing the cost for a sort merge-join) or the multiblock read count (improving the performance of full table scans over index scans).

4.5 Sort-/Hash-based Algorithms

Many query-processing operations (intersection, join, duplicate elimination) can be implemented using sort- or hash-based algorithms and there exist many dualities between the two types of algorithms.

4.5.1 Sort-based algorithms

Merge join sorts the relations first by the join attribute, so that interleaved linear scans will encounter these sets at the same time. Taking advantage of a pre-existing ordering in one or both relations is called interesting order. Aggregation can be done early while sorting/merging (combining duplicates whenever they are found), therefore reducing the output size.

For sorting, an external merge sort is often used. In the sorting phase, chunks of data small enough to fit in main memory (M items) are read, sorted, and written out to a temporary file on disk. In the merge phase, the sorted subfiles are combined into a single larger file with a recursive $\frac{M}{B}$-way merge: B elements from each sorted list are loaded into internal memory, and the minimum is repeatedly outputted. There can be additional passes for merging where in a first pass, the sorted chunks are merged into larger sorted chunks, which are again merged, etc...

4.5.2 Hash-based algorithms

In classic hash join, an in-memory hash table is built using one input (typically the smaller) hashed on the join attribute. Tuples from the other input are hashed and the hash table is probed for matches.

Problems occur when the smaller relation does not fit into memory. The grace hash join algorithm solves this by partitioning both input relations via a hash function and joining these partitions (with the same hash function). This is further improved with hybrid hash join which does not write all the partitions to disk, but keeps as much partitions as possible from the first relation in memory, avoiding unnecessary disk reads if memory is large enough. With partitioned hash join, the chunks are of cache size, resulting in no more cache misses during the join. Because there can be too many partitions this way (more than TLB-entries, causing TLB misses),

there is multi-pass radix partitioning which does the partitioning in multiple passes. In parallel radix join, the partitioning is parallelized among threads.

4.5.3 Duality

Aspect	Sorting	Hashing
In-memory algorithm	Quicksort	Classic Hash
Divide-and-conquer paradigm	Physical division, logical combination	Logical division, physical combination
Large inputs	Single-level merge	Partitioning into overflow files
I/O Pattern	Sequential write, random read	Random write, sequential read
I/O Optimization	Read-ahead, forecasting	Write-behind
	Multi-level merge	Recursive overflow resolution
Very large inputs	Number of merge levels	Recursion depth
	Non-optimal final fan-in	Non-optimal hash table size
Optimizations	Merge optimizations	Bucket tuning
Better use of memory	Reverse runs & LRU	Hybrid hash
	Replacement selection	Single input in memory
Aggregation	Aggregation in replacement selection	Aggregation in hash table
Interesting orderings	Merge-Join without sorting	N-way joins, hash-merging

In sort-based algorithms, a data set is divided into subsets using a physical rule (chunks as large as memory) and combined using a logical step (merging). In hash-based algorithms, a logical rule is used for division (hash values), but combination is a physical step (concatenation). The I/O pattern of sort-based algorithms is random during merging whereas it is random during partitioning for hash-based algorithms. Bucket tuning deals with using memory more efficiently in hash-based algorithms, in sort-based algorithms there are optimisations for the merge process (using maximal fan-in for the final merge). An equivalent to hybrid hashing for sorting is to retain some or all of the pages of the last run written in memory, which can be done particularly easily if the runs are written in reverse order and scanned backwards. With replacement selection (using a priority heap), larger runs can be generated. The equivalent to interesting orderings for hashing is to partition N relations and join their partitions.

In terms of performance, merge-join is inferior for unsorted inputs of very different size. But for hash join, the smaller relation should be predictable. Merge join is less affected by skewed data and can outperform hybrid hash join. Therefore, neither the input size nor the memory size determines the choice between sort- and hash-based query-processing, but the relative sizes, the danger of nonuniform hash value distributions and the opportunities to exploit interesting orderings.

4.6 Operators

The fastest way to access tuples is by using its row id, which can be found in the query itself (primary key) or through an index (either on a primary key that is different than the row id or with predicates over indexed attributes). Row IDs can also be used when only part of a predicate is indexed (by retrieving those row IDs that match this part, getting the tuples, and checking the rest of the predicates on them). When there are multiple indices, the list of row IDs can be intersected, or one can be used for fetching the tuples by the row ID and then checking them against the second predicate.

Full table scans are the upper bound in cost for retrieving data, but not necessarily the slowest method (can be faster than very inefficient plans or indices with low selectivities). For self joins (e.g. `WHERE T.a > T.b`) and when sequential access is better than many random accesses, it can be preferable. It is possible to minimize the overhead of scans by using shared scans (using the cursor from another query), sample scans (only reading samples if the query/workload supports this), or by using column stores (as scanning a compressed column using SIMD can be fast).

With clustered indices, data is organized according to the index (for B-trees it is sorted, for hash indices, tuples with the same key are on the same block), which can speed up retrieval (e.g. allowing sequential scanning of blocks). Table scans using an index can be expensive but will return sorted data, so it might be overall less expensive. Furthermore, scans in memory can be reasonably fast.

There are also special operators tailored to particular data types which need to be considered by the query optimizer. Nowadays, the emphasis is often on using the caches efficiently, exploiting SIMD, and taking advantage of the hardware (e.g. by prefetching into the caches).

4.6.1 Sorting and Aggregation

Sorting is important and often performed, either because output is required sorted or because some operations (distinct values, aggregations) are easier over sorted data. But sorting requires extra space. If data does not fit in main memory or when many queries run at the same time sharing memory (therefore decreasing the space per query), external sort algorithms (see above) are used. Prefetching/double buffering can speed up sorting and we can take advantages of indices (ideally clustered with sequential access).

An alternative is to use hashing for some operations (distinct values, aggregations) by building hash tables on the attributes of interest. As with sorting, external hashing can be used where data is (repeatedly) partitioned using a hash function.

4.6.2 Joins

Nested loop joins have complexity of $\mathcal{O}(|R| \times |S|)$, but are still used in practice. The outer table should be the smaller table to maximize the sequential access in the inner loop. In block nested loop joins, the blocks of the inner table are only read once for every block of the outer table, therefore reducing the number of reads by the number of tuples per block (of the outer table). For selective joins, having an index on the inner table can reduce I/O a lot. Zone maps can also be used in combination with nested loop joins to determine whether a block of the inner table can be skipped. On a sorted inner table, we only need to scan until no more matches are possible and can employ binary search. If both tables are sorted, a sort-merge join can be used where we do not go back on any of the tables.

4.7 Distributed Query Processing

Distributed databases are used because of cost and scalability (scaling out) and integration reasons (different software modules/legacy systems). A parallel database system is a particular type of distributed system, but there are others (e.g. heterogeneous distributed systems).

4.7.1 Basic Approach

In a distributed system, the query rewriter also selects the partitions of a table that must be considered to answer a query. The optimizer must also decide at which site each operation is to be executed. In the query plan, the send and receive operators encapsulate all the communication activity so that all other operators can be implemented and used in the same way as in a centralized system. The database catalog has to maintain the partitioning schema (what tables have been partitioned/how can they be reconstructed) and physical information such as the location of partitions. In a distributed system, the catalog can be stored at one central site or replicated at several sites. Caching can be employed to reduce communication costs. If the catalog data becomes too big, it can be partitioned.

In the dynamic programming approach for enumeration, plans can only be pruned if there exists another plan that does the same or more work and

the cost at all sites (including shipping) that may be involved in the query is lower. We distinguish between deep plans in which every join involves at least one base table and bushy plans where a join can also the result of two other join operations. In distributed systems, the cost model also needs to consider communication costs. The costs of machines/links can be weighted to incorporate slower machines/links. The classic cost model minimizes resource consumption, but not response time for distributed systems because of intraquery parallelism: A plan with higher resource consumption may be faster because some parts of the query can be executed in parallel at different nodes. There are alternative cost models that incorporate pipelined and independent parallelism. Traditional query execution optimizations/techniques can be naturally applied in a distributed system with send/receive operators, but there also specialized optimizations:

1. **Row Blocking**: Tuples are shipped in a blockwise fashion to reduce the network overhead. This also compensates for burstiness in the arrival of data.
2. **Optimization of Multicasts**: Depending on the network setup, forwarding data over an intermediate node (instead of broadcasting all the data from an indivdual node) may be beneficial.
3. **Multithreaded Query Execution**: Establishing several threads at a site can be beneficial, but not in all scenarios (it can also cause additional load on the disk/resource contention, reducing the overall performance).
4. **Joins with Horizontally Partitioned Data**: If the tables are horizontally partitioned, joins may be processed in a number of different ways (e.g. first merging the partitions and then joining or the other way). Furthermore, it sometimes can be deduced that the join of some partitions will be empty.
5. **Semijoins**: The idea of a semijoin is to send only the column(s) of A that are needed to evaluate the join predicates from site 1 to site 2, find the tuples of B that qualify the join at site 2, send those tuples to site 1 and then match A with those B tuples at site 1. The optimizer may use this variant based on the cardinalities of the tables, the selectivity of the predicates and the location of the data used in other parts of the query.
6. **Double-Pipelined Hash Joins**: To execute a join, two hash tables (for both inputs) that are initially empty are constructed. When a tuple arrives, the hash table of the other input is probed for matches

(which are immediately output) and the tuple is added to the hash table of the input.

7. **Pointer-Based Joins and Distributed Object Assembly**: In some systems, foreign keys are implemented by explicit references that contain the address of an object (the site and storage location). Grouping these references by the site when joining allows fetching them in one batch. The P(PM)*M algorithm is based on this idea, but also ensures that the tuples are in the same order as before after the join. If a query involves several pointer-based joins, object assembly algorithms can be used: The grouping is performed multiple times and round-trips are minimized by fetching as much referenced objects at a site as possible.

8. **Top N and Bottom N Queries**: The number of tuples that are shipped at every site can be reduced with different techniques (e.g. for the overall top 10 tuples, every site needs to only ship its top 10 tuples).

4.7.2 Client-Server Systems

We distinguish between peer-to-peer (every site can act as a server and as a client), (strict) client-server (every site has the fixed role of client or server; often no interactions among clients and among servers), and middleware/multitier (hierarchical organization where every site is the server for the sites at the upper level and the client for the lower-level sites).

In these systems, queries can be executed at client machines that initiate the queries or at the server machines that store the relevant data. In query shipping, queries are executed at servers. The server evaluates the query and ships the result back to the client. In systems with several servers, there needs to be a middle-tier that carries out joins or gateways between the servers. In data shipping, queries are executed at the client machine and data is cached at client machines in main memory or on disk. The scan operators at the client uses the cached pages and faults in all the pages that are not cached. Hybrid shipping provides the flexibility to execute query operators on client and server machines and allows the caching of data by clients. Query shipping generally performs well if the server machines are powerful but the scaling can be bad. Data shipping scales well but communication costs can be high if caching is not effective. Hybrid shipping can combine the best of both worlds, but query optimization can be more complex because more options must be considered. For instance, sometimes it is better to read data from the servers'

disk even if it is cached and sometimes the best strategy is to ship cached base data/intermediate query results from the client to a server.

In client-server systems, the optimizer needs further select the side (where the options are given by the supported shipping methods). If there is replication, translating a server annotation for a scan involves selecting one specific server machine (e.g. based on heuristics or in a cost-based manner). Queries can be optimized at the client, server, or both (e.g. parsing/rewrite at the client, optimization/plan refinement at the server). The system that performs the optimization needs to guess the state of the network/other servers based on statistics of the past or discover it with additional messages. Another question is when to optimize a query. This can be done at application compile time, but then the plan cannot adapt to changes such as shifts in the load of sites. An alternative is to generate several alternative plans and choose the plan/subplans just before executing the query. Two-step optimization generates a plan that specifies the join order, join methods, and access paths at compile time. Just before the query is executed, the plan is transformed and site selection is performed. This can be used to exploit caching by placing query operators at a client if the data is cached. However, it can result in plans with unnecessarily high communication costs (when some join orders would be better because of the data distribution and the first step of the optimization was carried out ignoring the location of data).

For query execution, a problem in hybrid shipping systems is how to deal with transactions that first update data in a client's cache and then execute a query at a server that involves this updated data. Solutions are to propagate all relevant updates or to pad the results returned by the server at the client using the updated data.

4.7.3 Heterogeneous Database Systems

When dealing with heterogeneous systems, a challenge is to find query plans that exploit the specific capabilities of every database in the best possible way. Another challenge is semantic heterogeneity (e.g. different currencies). Furthermore, every database has its own API and decides autonomously when/how to execute a query and might not be designed for interaction with other systems. The general goal is to encapsulate the heterogeneity of the component databases and use existing homogeneous distributed database technology as much as possible.

An approach to address these challenges is the wrapper architecture. There is a mediator that clients connect to. It parses a query, performs optimization and executes some of the operations. It also maintains a

global schema and an external schema with information on which parts of the global schema are stored where. A wrapper/adaptor is associated to every database. The wrapper provides a set of planning functions which are called by the optimizer to construct subplans/wrapper plans. Depending on the system that is behind a wrapper, only some specialized access methods (e.g. predicates on only some attributes) can be provided and some access methods can be emulated/executed by the mediator (e.g. filtering of the results or joins regardless where the tables are stored).

For the cost estimation, there are three approaches. In the calibration approach, a generic cost model is adjusted using a set of test queries. In the individual wrapper cost model approach, wrapper developers provide a separate cost model for every operation. In the learning curve approach, the cost of wrapper plans is estimated based on execution statistics. This allows automatic and dynamic adaptation to changes in the system.

For query execution, there are some specialized techniques. Bindings simulate a nested-loop join: The mediator asks the wrapper of a database to evaluate a `WHERE joinattr=?` query for every tuple of the other join relation. A block of tuples can be passed (if this is supported by the wrapper), which roughly corresponds to a block-wise nested loop join. The idea of cursor caching is to optimize a query only once in order to reduce the overhead of submitting the same query.

4.7.4 Dynamic Data Placement

Replication and caching share the same goals (reduce communication costs/balance system load), but there are some differences. Replication takes effect at server machines, is typically coarse-grained, replication decisions are usually more long-term, replication does affect the database catalog, it makes use of propagation-based protocols (whereas caching uses invalidation-based protocols), replicas are established by a separate process (whereas caching is a by-product of query execution), and replication can be used to improve system reliabilty. Caching and replication are complementary techniques that both should be implemented.

Replication algorithms can be classified into algorithms that try to reduce communication costs by moving copies of data to servers that are located near clients that use the data and algorithms that try to replicate hot data in order to balance the load. The ADR algorithm is targeted to reduce communication costs. It is based on the idea that the nodes that replicate an object should form a connected (sub)graph and it therefore expands/-contracts the nodes that replicate an object at the borders.

The cache investment algorithm is based on the idea of what-if analyses. The cost (investment) and benefits of caching parts of a table or index are calculated. The query optimizer is extended such that it decides to execute queries at clients if the query involves data that should be cached at these clients. It may therefore decide to execute suboptimal plans if the execution of them brings relevant data into the client's cache.

Besides base data, systems can also cache views (materialized views), which is more complex to implement. Keeping the views consistent can be hard and deciding what views to materialize is generally very difficult. Furthermore, query optimization becomes more complicated and expensive because the views must be considered.

4.7.5 New Architectures

In economic models (e.g. systems that process queries by carrying out auctions), every server tries to maximize its own profit by selling its services to clients. Dissemination-based systems rely on push technology and send data to clients before they ask for it. They often scale better than traditional request-driven systems.

Chapter 5

Transaction Management

Concurrency control is the activity of coordinating the actions of processes that operate in parallel, access shared data, and therefore potentially interfere with each other. Recovery is the activity of ensuring that software and hardware failures do not corrupt persistent data, i.e. changes from uncompleted transactions are not visible and those of committed transactions are recorded and visible. A transaction is an execution of a program that accesses a shared database (usually with the purpose of modifying the data, in contrast to queries which read data). The goal of concurrency control and recovery is to ensure that transactions execute atomically, meaning they access shared data without interfering with other transactions and if a transaction terminates normally, all of its effects are made permanent (and otherwise, it has no effect at all). The last operation of a transaction must be a commit or abort (the commit and begin of a transaction can be implicit/automatic in practice, e.g. sessions and autocommit). Transactions that are not yet committed or aborted are called active, we call them uncommitted if they are aborted or active. Savepoints allow to temporarily commit the changes made by a transaction and rollbacks to savepoint allow to cancel the changes made after a savepoint. Transactions may issue an abort themselves (because of errors) or the abort may be imposed on a transaction. Executing a transaction's commit constitutes a guarantee by the DBS that it will not abort the transaction and that its effects will survive subsequent failures of the system. As long as a transaction is active, the user cannot be sure that its output will be permanent. Therefore, commit is also important for read-only transactions (queries).

When a transaction aborts, its effects must be wiped out. The effects are effects on data (values that the transaction wrote) and effects on other transactions (transactions that read values written by the transaction). Aborting the affected transactions may trigger further abortions, which is

known as cascading aborts.

The lost update phenomenon occurs whenever two transactions, while attempting to modify a data item, both read the item's old value before either of them writes the item's new value. Inconsistent retrieval occurs whenever a retrieval transactions reads one data item before another transaction updates it and reads another data item after the same transaction has updated it. These phenomenon do not occur for serializable transactions.

We consider a database model consisting of a transaction manager (TM), scheduler, and a data manager (DM), which internally has two modules: A recovery manager (RM) and a cache manager (CM). Given an operation, the scheduler can execute the operation, refuse to process the operation (causing the transaction to abort), or delay the execution. The study of concurrency control techniques is the study of scheduler algorithms that attain serializability and either recoverability, cascadelessness, or strictness. The TM usually has a transaction table (list of active transactions), transaction handlers (pointer to the structures containing all the relevant information related to a transaction, e.g. the timestamp), a lock table, and a log. Log entries for the begin of a transaction are only created if they are requested explicitly. There are systems (e.g., Oracle) without lock tables where the locks are stored in the blocks containing the tuples, which reduces contention and the size of the lock table is no longer an issue. The headers in the blocks are also used to keep some metadata related to transactions, e.g. the timestamp of the latest change.

5.1 Serializability Theory

We use $r_i[x]$ / $w_i[x]$ to denote reads/writes by transaction T_i on data item x and $o_i[x]$ for an arbitrary operation (read or write). c_i / a_i is used to denote commit/abort operations. Arrows indicate the order in which operations execute (i.e. "happens before"/"happens after" relations). A transaction T_i is a partial order with ordering relation $<_i$ where:

1. $T_i \subseteq \{r_i[x], w_i[x]\} \cup \{a_i, c_i\}$
2. $a_i \in T_i$ iff $c_i \notin T_i$
3. If t is c_i or a_i, $p <_i t$ for any other operation $p \in T_i$
4. If $r_i[x], w_i[x] \in T_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$ (meaning the order of read/writes on common data items is specified)

Transactions can be represented as DAGs. We assume that each value written by a transaction depends on the values of all the data items that it previously read.

The concurrent execution (the order in which the operations were executed relative to each other) of transactions is modelled by histories. A complete history H over transactions $T = \{T_1, \ldots, T_n\}$ is a partial order with ordering relation $<_H$ and:

1. $H = \cup_{i=1}^{n} T_i$ (same operations as the transactions)
2. $<_H \supseteq \cup_{i=1}^{n} <_i$ (execution honors operation orderings within T)
3. For any two conflicting operations (same data item and at least one of the operations is a write, or aborts of a transaction that updated a data item with a read/write of another transaction) $p, q \in H$, either $p <_H q$ or $q <_H p$.

A history is a prefix of a complete history and therefore represents a possibly incomplete execution of transactions. Given a history H, the committed projection $C(H)$ of H is the history obtained from H by deleting all operations that do not belong to transactions committed in H.

Two histories are equivalent if they are defined over the same set of transactions and have the same operations; and they order conflicting operations of non-aborted transactions in the same way (committed transactions see the same state and leave the database in the same state).

5.1.1 Serializability

An execution is serializable if it produces the same output and has the same effect on the database as some serial execution of the same transactions. Serializability is the canonical definition of correctness for concurrency control in DBSs. Serializable executions preserve database consistency (according to some consistency predicates defined by the database designer), as long as transactions are correct. However, the DBS may execute transactions in any order, as long as the effect is the same as that of some serial order. If execution in a particular order is needed, this must be ensured by mechanisms outside the DBS. Critical sections can be considered as a type of transaction, then mutual exclusion (which is more appropriate for some applications, e.g. non-terminating ones) is a strong form of serializability.

A history H is serializable (SR) if its committed projection $C(H)$ is equivalent to a serial history (without interleavings) H_s. This can be checked by examining the serialization graph of H, denoted $SG(H)$. This is a directed

graph whose nodes are the transactions in T that are committed in H and whose edges are alle $T_i \rightarrow T_j$ such that one of T_i's operations precedes and conflicts with one of T_j's operations in H. A single edge can be present because of more than one pair of conflicting operations and the relation is generally not transitive. The serializability theorem states: A history H is serializable iff $SG(H)$ is acyclic. Furthermore, H is equivalent to any serial history that is a topological sort of $SG(H)$.

5.1.2 Recoverability

Recoverability is required to ensure that aborting a transaction does not change the semantics of committed transactions' operations. Enforcing recoverability does not remove the possibility of cascading aborts. But even without cascading aborts, we cannot simply implement abort by restoring the before images of all writes of a transaction. This is only possible for strict executions.

A transaction T_i reads x from T_j in history H if:

1. $w_j[x] < r_i[x]$
2. $a_j \not< r_i[x]$
3. If there is some $w_k[x]$ such that $w_j[x] < w_k[x] < r_i[x]$, then $a_k < r_i[x]$

Based on that, we can classify histories into:

- **Recoverable (RC)**: Whenever T_i reads from T_j in H and $c_i \in H$, $c_j < c_i$ (each transaction commits after the commit of all transactions from which it read, i.e. transactions commit in their serialization order)
- **Avoids Cascading Aborts (ACA)**: Whenever T_i reads x from T_j, $c_j < r_i[x]$ (transactions may read only those values that are written by committed transaction or itself)
- **Strict (ST)**: Whenever $w_j[x] < o_i[x]$, either $a_j < o_i[x]$ or $c_j < o_i[x]$ (no data item may be read or overwritten until the transaction that previously wrote into it terminates, either by aborting or committing)

We have:

$$ST \subset ACA \subset RC$$

SR intersects all of the sets RC, ACA, and ST, but is incomparable to each of them. DBS that must correctly handle transaction and system failures must therefore enforce (at least) recoverability in addition to serializability.

Properties of histories are called prefix commit-closed if, whenever the property is true for history H, it is also true for history $C(H')$ for any prefix H' of H. RC, ACA, ST, and SR are prefix commit-closed.

These definitions can easily be extended beyond read and writes by simply defining which of the new operations conflict with each other (meaning the computational effect depends on the order).

An alternate (weaker) definition for equivalence and therefore serializability is given by view equivalence/serializability. Producing an efficient scheduler that produces exactly the set of all view serializable histories is only possible if $P = NP$.

Besides this classification, there are also the ANSI isolation levels:

- Read uncommitted: Dirty reads (reads of uncommitted data) allowed, not ACA and allows non-recoverable executions.
- Read committed: No dirty reads, but a read of the same item at different times can result in different values (non-repeatable read). The histories are ACA but might not be serializable and strict.
- Repeatable read: No dirty reads, non-repeatable reads, but phantom reads can occur. The histories are also ACA but might not be serializable and strict.
- Serializable: No dirty reads, non-repeatable reads, and phantom reads. Does not match the canonical definition of serializability.

These isolation levels do not include write-write conflicts (because they focus solely on isolation), serializable histories according to the SQL isolation level might therefore be not serializable according to the canonical definition. However, the guarantees provided by locking protocols are usually stronger.

5.2 Two Phase Locking

We generally distinguish between aggressive and conservative schedulers. Aggressive schedulers tend to avoid delaying operations, at the expense of possibly having to abort more transactions. Conservative schedulers tend to delay operations. Their relative performance differs depending on the application/transaction workload (aggressive schedulers are generally good when conflicts occur rarely). Generally, conservative schedulers try to anticipate the behaviour of transactions (i.e. their read- and writesets), in the extreme case by letting them predeclare their read-/writeset.

In the basic two phase locking (which is the canonical implementation of concurrency control), we have read locks and write locks where the locks conflict if the operations are of conflicting type. It works as follows:

- When the scheduler receives an operation, it tests if the lock for the data item conflict with an already set lock. If so, it delays the operation, forcing the transaction to wait until the lock is free again.
- Once a lock has been set, it may not be released until the operation of the lock has been processed.
- Once the scheduler has released a lock for a transaction, it may not subsequently obtain any more locks for that transaction (on any data item). This is known as two phase rule because each transaction may be divided into two phases: A growing phase (when locks are obtained and/or upgraded) and a shrinking phase (when locks are released). This rule ensures that all pairs of conflicting operations are scheduled in the same order, not only individual ones.

Transaction aborts due to concurrency control issues are avoided by this scheme and the histories are conflict serializable. Weaker SQL isolation levels can be achieved by modifications (uncommitted read by not acquiring locks before reading, read committed by releasing locks directly after reading). However, 2PL schedulers are subject to deadlocks. One strategy is timeout, i.e. aborting transactions that have been waiting too long (which is a system parameter, balancing response time and throughput) for a lock. This can lead to false positives, but the mechanism also deals with failed clients (e.g., because of a network problem). Another approach is to detect the deadlocks by checking for cycles in the waits-for-graph (although this is rarely done in practice as bulding the graph is too expensive). When a deadlock is detected, it must be decided which transaction is the victim (i.e. aborted). There are are many factors to consider when choosing the victim (lowest abortion cost, amount of effort that has been invested in the transaction, amount of effort for finishing the transaction, number of cycles that contain the transaction, number of times the transaction was already restarted, ...).

Conservative 2PL avoids deadlocks by requiring each transaction to obtain all of its locks before any of its operations are submitted (i.e. predeclaration of the read-/writeset). If there is at least one conflict, no lock is granted.

Strict 2PL requires the scheduler to release all of a transactions's locks together (after the commit/abort acknowledgement). The scheduler can generally only be sure that no more operations arrive after the commit/abort

acknowledgement, so strict 2PL is sensible from a practical point of view. Furthermore, it guarantees a strict execution (for that it is theoretically only necessary to hold write locks until after a transaction commits/aborts) and almost all engines implement strict 2PL.

The lock table is usually implemented as a hash table and to make releasing locks of a transaction fast, the locks of a transaction are often linked together in the lock table. It can be partitioned such that the access to it does not become a bottleneck. The table should be large enough to avoid conflicts. New requests are usually inserted at the end of the linked list (for the item). When read lock requests are allowed to continually jump ahead of write lock requests, write requests can be postponed indefinitely. This can be solved by servicing the queue first-come-first-served or only allowing read lock request to jump ahead if the write lock request has not been waiting too long. Lock, unlock, read, and write must be implemented atomically in order for 2PL to be correct. Depending on the granularity of read/writes (blocks, records, ...), blocks need to be locked while a write is being applied to a record contained in that block to avoid overwriting of different records when writing back a block. The phantom problem can be solved by locking additional control information (e.g. EOF), index locking, or predicate locking.

By providing additional atomic operations such as increment and decrement (which commute and therefore can set weaker locks than write operations), the concurrency for hot spots can be increased. This requires atomic implementation of these operations (e.g. using an internal lock during the execution) and the additional operation types can then be implemented by introducing lock types for the new operation types and defining which lock types conflict (where two lock types conflict if the operation types do not commute).

Multi-Granularity Locking (MGL) allows each transactions to lock coarse granules (e.g. for longer transactions that access many items) or at a finer granularity. Two transactions must be prevented from setting locks on two granules that overlap. This is accomplished by introducing intention locks. Before setting the lock on a data item, the intention lock is set on all ancestors (with a coarser granularity, containing the data item). This is not possible if one of the ancestors is explicitly locked (for writing) by another transaction. The MGL protocol works as follows:

1. If x is not the root, to set the read or intention read lock of x, T_i must have an intention read or intention write lock on x's parent.
2. If x is not the root, to set the write or intention write lock of x, T_i must have an intention write lock on x's parent

3. To read (or write) x, T_i must own a read or write (or write) lock on some ancestor of x (or x itself)
4. A transaction may not release an intention lock on a data item x if it is currently holding a lock on any child of x (i.e. locks should be released in leaf-to-root order)

When transactions start locking items of fine granularity, the scheduler may start requesting locks at the next higher level of granularity when a certain number of locks is reached, which is called lock escalation. In some cases, aborting and restarting a transaction may be less expensive than lock escalation (which may cause a deadlock). MGL can be extended to generalized locking graphs (not only trees) where transactions need intention write locks on all parents of x for writes.

Cursors can also be used as a way to implement concurrency control (often with weaker guarantees). As the cursor advances, the corresponding locks are obtained on the next tuple and released on the previous. Cursors are not allowed to get ahead of each other (therefore preventing phantoms). This is a way to organize locking in a sequential way and useful for select statements scanning a table and using a complex predicate to decide what to update.

Furthermore, locks can be converted (e.g., read to write or tuple to whole table after too many locks have been requested), which can lead to deadlocks if not implemented with care. SQL Server provides the following locks:

- Shared locks (reads)
- Update locks (updates, but not writes)
- Exclusive locks (writes)
- Intent locks
- Schema locks (preventing schema changes)
- Key range locks (to prevent phantoms)

2PL can also be used in distributed settings. With strict 2PL, each local 2PL scheduler has all the information for processing operations, without communication with the other sites. However, it is possible that there are distributed deadlocks that are not detected by checking only the local wait-for-graphs. A global deadlock detector receives all WFGs, checks for distributed deadlocks, and selects the victim (based on additional information that can be sent along the local WFGs). Because most cycles are short, an alternative, potentially more efficient, technique is path pushing

where sites exchange paths in their WFGs among each other and build up the global WFG in this way. Besides these detection schemes, there are prevention schemes where transaction are aborted when a deadlock might occur. Such a scheme can be priority based, with the special case of timestamp-based deadlock prevention which uses timestamps as priorities (to avoid livelocks which can occur in a simple priority-based system). Each transaction is assigned a timestamp and the older a transaction, the higher its priority. A transaction is then only allowed to wait if it has a higher priority than the other transaction.

Locking can cause thrashing at certain point (decrease in throughput) because of data contention. Because deadlocks are generally much rarer than conflicts, this is often caused by waiting in lock queues. With regards to granularity, throughput initially (when using a finer granularity) shrinks, then increases, and then starts to decrease at a certain point again. This is caused by locking overhead, data contention, and resource contention. However, a given system may not see the entire granularity curve. For long transactions, a very coarse granularity may be the best option.

If data items are structured as nodes of a tree and transactions always access data items by following paths in the tree, this access behavior can exploited by using locking protocols that are not two phase, using techniques such as lock coupling and B^{link}-trees (see Section 3.2.1).

5.3 Snapshot Isolation

Snapshot isolation is a form of multiversion concurrency control with a different definition of conflicts. It can be implemented without locks, but locks are often still used (e.g., after too many aborts/for long running transactions). Reads will only access items that where committed as of the time the transaction started. Writes are carried out in a separate buffer and become only visible after a commit. On commit, the engine checks for conflicts with the first-committer-wins rule: T_1 with timestamp t_1 is aborted if there is a T_2 such that T_2 committed after T_1 started and before T_1 commits and both transactions updated the same object. Some systems also check earlier for conflicts such that transactions do not run for longer when they cannot commit anyway. Snapshot isolation does not result in conflict serializable histories, the correctness does depend on the application semantics (some constraints can be violated, but there are other ways such as triggers to enforce those). It provides "Read Consistency" isolation which is stronger than the ANSI SQL isolation levels (not only are all reads clean and there are no phantoms, but the reads return

also the values per the transactions start). To implement snapshot isolation, a LSN is added to the tuples and writes create a new copy of the tuple and check at the end whether the latest version of all written tuples is the one written. Oracle implements it by keeping undo records in an undo segment. When a block is too new (higher system change number), a copy is made and undo records are applied to recreate a version with a suitable SCN. When the undo records are kept for longer (or are made persistent), the user can perform flashback queries, i.e. run a query on an old database state.

5.4 Recovery

The most important types of failures are transaction failures (aborts of transactions, requires undo of changes), system failures (loss or corruption of volatile storage, e.g. because the power fails), and media failures (destruction of any part of stable storage). The techniques for dealing with media failures are conceptually similar to those used to handle system failures. In both cases, a redundant copy is kept in another part of storage (in the case of media failures replication is used and data files can be separated from log files).

The committed database state with respect to a given execution is the state in which each data item contains its last committed value. The goal of restart is to restore the database into its committed state with respect to the execution up to the system failure such that it is in a consistent state again. This can be done using only information saved in stable storage. The RM (recovery manager) partially controls the CM's (cache manager's) flush operations to ensure that stable storage always has the data that the RM needs to process a restart correctly. We assume that writes are atomic (either execute in their entirety or not at all). The granularity of data items that are parameters to writes to the DM (e.g. records) may be different from that which can be atomically written to stable storage (e.g. blocks), which needs special attention when dealing with recovery algorithms.

DMs that keep exactly one copy of each data item in stable storage perform in-place updating (always destroy the old value), whereas with shadowing, multiple older versions called shadow copies are kept. With shadowing, there is often a directory with one entry per data item which points to the stable storage location. A cache slot usually contains a dirty bit (set if and only if the value of the item is different from the value in stable storage) besides the actual value. The point in time when flushing happens is often a decision left to the RM and different RM algorithms use different strategies. While a cache slot is pinned, it is never flushed. We assume the

scheduler invokes RM operations in an order that produces a serializable and strict execution, allowing restoring with before images. Because the stable database might contain values written by uncommitted transactions or might not contain values written by committed ones, the RM usually stores additional information in a log, which is a representation of the history of execution. Updates to it go directly to the stable storage device and must be acknowledged (there are also designs with an in memory log and flushing to a persistent log). In a physical log, information about the values of data items written by transactions (including the order) is stored. A log may also contain descriptions of higher level operations, which is called logical logging (can require fewer log entries, but increases the complexity when restarting). Log records are stored in log blocks which are often much smaller than data block (e.g., 512 bytes which is the size of a physical sector). Besides undo related information (before images), redo related information (change vectors describing changes to blocks of data, i.e. multiple after images because an update can cause modifications in multiple blocks), pointers to other log records, and transaction IDs, systems often log a LSN and SCN. The log sequence number is used to order transactions whereas the system change number timestamps events (i.e. indicates the version of data). In addition to the log, the RM may also keep in stable storage the active, commit, and abort lists (often stored as part of the log). In many RM algorithms, the transaction is committed iff its transaction identifier is in the commit list.

Oracle uses a circular redo log buffer that is occasionally flushed by a log writer (i.e. in contrast to the following algorithm descriptions, log records do not end up immediately on disc in practice). Several log files are used and written to/archived in a circular manner, allowing parallel writing and archival. Systems often commit transactions in groups or batches (group commit) to avoid frequent flushes of the log buffer. The most common implementation for managing the log is write ahead logging: There is separate persistent storage for data and the log. The log records corresponding to a change must be written to the log before changes to the date in the buffer cache are flushed to permanent storage (the reason for the "write ahead" in the name).

ARIES style recovery refers to a canonical way to implement recovery with three phases. First, the latest completed checkpoint in the log is searched and the log is traversed from it to the end to identify the active transactions and dirty pages at the time of the crash (analysis phase). In the redo phase, all updates are applied, starting from the log entry matching the lowest SCN in the dirty pages (as the checkpoint might have written dirty pages, this can be before the checkpoint). In the undo phase,

all transactions that were active at the time of the crash are undone.

Nowadays, SSDs or even NVM is often used for logs. This changes quite a few things (no more blocks, byte-addressable), especially with non-volatile memory, as flushing upon commit is no longer necessary and it is possible to keep checkpoints in the NVM.

5.4.1 Undo/Redo

An RM requires undo if it allows an uncommitted transaction to record in the stable database values it wrote (these effects must be undone by restart). On the other hand, an RM requires redo if it allows a transaction to commit before all the values it wrote have been recorded in the stable database (these effects must be redone). By regulating the order of a transactions commitment relative to writing its values in the stable database, an RM can control whether it requires undo or redo and we can therefore classify the algorithms into four categories (described below) which are also known as Steal/No-Force (undo and redo), Steal/-Force (undo but not redo), No-Steal/No-Force (redo but not undo), and No-Steal/Force (neither undo nor redo). The terms steal and force are from the buffer cache perspective where steal means that an uncommitted transaction is allowed to overwrite in persistent storage the changes of a committed transaction and force that all changes made by a transaction must be in persistent storage before it commits. All RM implementations must observe the following two design rules:

- **Undo Rule** (Write Ahead Log Protocol): If x's location in the stable database presently contains the last committed value of x, this value must be saved in stable storage before being overwritten in the stable database by an uncommitted value.
- **Redo Rule**: Before a transaction can commit, the value it wrote for each data item must be in stable storage.

Recycling space in the log occupied by unnecessary information is called garbage collection. If the stable database copy of a data item does not contain its last committed value, restart must be able to find that value in the log. Therefore, an entry $[T_i, x, v]$ can be removed from the log iff T_i has aborted or T_i has committed but some other committed transaction wrote into x after T_i did (meaning v is not the last committed value of x). Note that even if v is the last committed value of x and stored in the stable database copy, the log entry cannot be deleted as a subsequent transaction can still overwrite the value and abort. But if the RM does not require undo, this cannot occur, so in this case the entry can also be

removed if v is the last committed value of x, the value in the database, and $[T_i, x, v]$ is the only log entry for x. The last condition ensures that some earlier log entries are not misinterpreted as x's last committed value.

As restart can be interrupted, it needs to be idempotent, i.e. it needs to produce the same result in the stable database as if the first execution had run to completion after an interruption.

Undo/Redo Algorithm

The undo/redo algorithm is the most complicated algorithm, but is very flexible because it leaves the decision when to flush almost entirely to the CM and is commonly used. It is geared to maximize efficiency during normal operation, at the expense of less efficient processing of failures. On a write, $[T_i, x, v]$ is appended to the log before writing the value into the cache slot. Reads simply read the data from the cache. On commit, the transaction is added to the commit list (at which point it is declared committed). On abort, the before image of all data items that were updated is copied into the cache and the transaction is added to the abort list. The restart procedure starts with the last entry in the logs and scans backwards toward the beginning. Two sets `redone` and `undone` track which data items have been restored by a redo / undo action. Until there are no more log entries or `redone` $\cup$ `undone` equals the set of all data items, the procedure performs for each entry $[T_i, x, v]$ if $x \notin$ `redone` $\cup$ `undone`:

- If T_i is in the commit list, copy v into x's cache slot and append x to `redone`
- Otherwise (i.e. T_i is in the abort or active list), copy the before image of x w.r.t. T_i into x's cache slot and append x to `undone`

Because of the backwards scanning, an update is only redone/undone if no other committed transaction subsequently updated x.

The restart procedure may have to examine every record ever written, which is addressed by checkpointing. In general, checkpointing marks the log, commit list, and abort list to indicate which updates are already written/undone in the stable database (such that restart knows they do not have to be undone/redone again). Optionally, it can also write the after images of committed updates or before images of aborted updates in the stable database. In commit consistent checkpointing, transaction processing is periodically stopped, all dirty cache slots are flushed and the end of the log is marked. Restart only scans until it reaches the last checkpoint marker (but may still have to examine older log records for certain before images). In cache consistent checkpointing, active transactions are only

blocked (i.e. no waiting until they finish). Then, restart only needs to redo the updates of transactions after the last checkpoint and undo the updates of the transactions that were active during the checkpoint/aborted later. In fuzzy checkpointing, only the dirty slots that have not been flushed since before the previous checkpoint are flushed (with the hope that most slots are flushed between two checkpoints, in which case the delay because of checkpointing is reduced), which guarantees that all updates of committed transactions that occurred before the penultimate (second to last) checkpoint have been applied to the stable database. For all checkpointing schemes, increasing the maximum length of time that a cache slot can remain dirty before being flushed increases the work of checkpoint and restart, but speeds up the system during normal operation.

Instead of logging whole data items (e.g. pages), it can be much more efficient to log only partial data items (the portion of each data item that was actually modified). The partial data item logging algorithm uses fuzzy checkpointing and logs partial data items. Log entries are written into a log buffer that is regularly flushed to the stable log. In this scenario, it can be worthwhile to deliberately delay commits (delayed commits/group commits) such that the buffer can fill up. With log buffering, the undo rule no longer holds. This can be solved by adding the log sequence number (LSN) to each cache slot. The CM then ensures that all log entries up to and including the one whose LSN equals the cache slots's LSN have been appended to the log before flushing a slot. In this implementation, update records contain the old/new value and the LSN of the previous update record of the same transaction (which allows efficient implementation of abort).

Before and after images can still consume too much space when using partial data logging. An alternative is to use logical logging. The RM needs then to consider more complex operations such as insertions, deletions, shifts within a page, etc... and a procedure for redoing/undoing these operations. Some undos or redos corresponding to logical log records may only be applicable to a data item when it is in exactly the same (logical) state as when the log was created. One solution for this is to write undo/redo procedures that have no effect when applied to a data item that is already in the appropriate state. Alternatively, a copy of the stable database state as of the last checkpoint can be kept (which is used to apply the logical updates). Another solution is to store LSNs in the data items, which allows restart to infer what updates already have been applied to a data item. Furthermore, with this LSN-based logging algorithm, restart can be more efficient by avoiding unnecessary undos and redos. Problems with logical logging can occur if the system fails when the database

does not contain all of the updates performed by a single logical operation, which can be solved by producing separate update records for each data item that is modified by a logical update. When LSN-based logging is used with LSNs on a page basis, it can be required to insert undo log records in order to have a LSN that represents the state of the page. An alternative is to store LSNs in records rather than pages, which does not require logging undos but does have extra space overhead.

The only forced I/O for undo/redo are log records and the buffer cache manager has a lot of freedom (I/O only triggered for block replacement).

Undo/No-Redo

To never require redo, the Undo/Redo algorithm only has to be modified such that all cache slots are flushed before committing (force). The restart procedure then only considers the undo part of the previously described procedure. One drawback of the approach is that for a hot spot data item, the required flush for every committed write may create a heavy I/O load.

With partial data item logging, update records do not need to include after images. The flushing on commit can be seen as a limited kind of a checkpoint and it can even be implemented by taking a cache consistent checkpoint just before adding a transaction to the commit list. If it is not implemented in this way, checkpointing is still needed (to ensure that restored before images of aborted transactions are eventually recorded in the stable database).

Although the log records can be smaller (with no after image), the trade-off does not pay off in practice as there is a write to the log with every update (I/O on data blocks which are often much larger). For this reason, undo/no-redo is rarely used in practice.

No-Undo/Redo

To never require undo, a new value is not recorded in the cache when an item is written, this only happens after a transaction commits. Read therefore has to check if the value was previously written by the transaction (this can for instance be done using intention lists which contain the after images of a transaction and can be indexed) and return this value if this is the case. Commit has to copy all after images into the cache. The restart algorithm only considers the redo part.

Similar to undo/no-redo, no before images are needed for partial data item logging. With this policy, there is forced I/O only on the log records and the

recovery procedure is shorter. The policy is similar to snapshot isolation and often used in systems that implement SI (e.g., Oracle).

No-Undo/No-Redo

To eliminate both undo and redo, all updates must be recorded in the stable database in a single atomic operation at the time the transaction commits. This can be achieved using a form of shadowing (shadow version algorithm/careful replacement algorithm) with two dictionaries (current/scratch dictionaries) and a pointer that indicates which dictionary is which (allowing swapping of them in one operation). With this approach, the scratch dictionary can be prepared and on commit, the dictionaries are swapped. However, only one commit at a time can be processed, accesses to stable storage are indirect, and the movement of data to new versions destroys data locality. The read procedure returns values from the scratch directory (if they differ from the current directory).

The policy is rarely used in conventional databases because indirection through the directory is expensive, it requires garbage collection, and moves data all the time (creating problems with the block representation, e.g., clustered indexes).

5.4.2 Media Failures

In contrast to system failures, the recovery mechanism for media failures must be designed with a probabilistic goal in mind (minimize the probability that all copies are destroyed). To increase resiliency, we generally want to keep the copies on devices with independent failure modes. Mirroring keeps a duplicate copy of each disk on-line (which increases the capacity for reads but not for writes, as writes are usually performed in sequence to protect against simultaneous failure). With archiving, the value of each data item is written to an archive database (similar to checkpointing) and the log can be used to bring the database back to its committed state. The difference is that the archive database needs to be updated to include updates that are only in the stable database and the cache (in contrast to checkpointing which only considers the cache).

A solution to alleviate the delay caused by checkpointing is to create an archive shadow directory (defining the state of the database at that time) when archive checkpointing begins. Alternatively, fuzzy checkpointing can be adapted for archiving. To reduce software complexity, it is often desirable to have the same restart algorithms for system and media failures.