Data Management Systems

{wnicole}@ethz.ch

ETH Zürich, HS 2020

This is a summary for the course *Data Management Systems* at ETH Zürich.
I do not guarantee correctness or completeness, nor is this document endorsed by the lecturers. Feel free to point out any erratas. It's probably smart to first reread your old DMDB summary!

Contents

1 Introduction

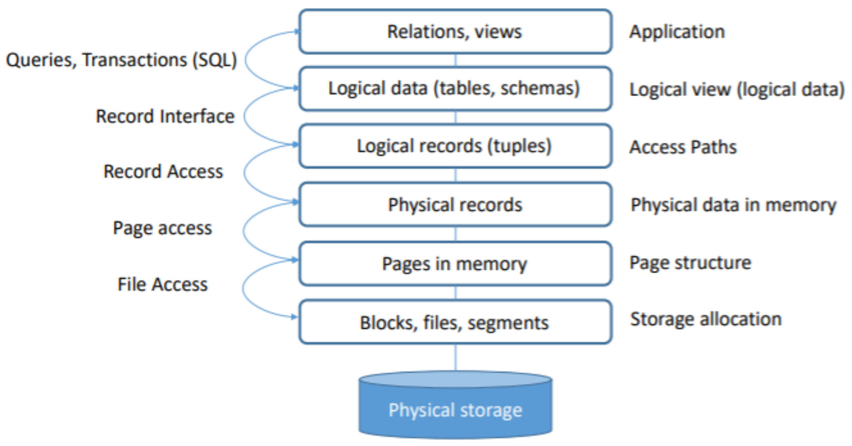


Figure 1: Architecture of a database.

Database Management System (DBMS) Tool that helps develop and run data-intensive applications. A DBMS pushes the complexity of dealing with the data (storage, processing, consistency) to the database rather than to the program.

Relational Database

Physical Data Independence The ability to modify the physical schema (to improve performance) without causing the application programs to be rewritten i.e. without affecting the conceptual / external view of the data. The database hides how the data is actually stored persistently and represented in memory.

Logical Data Independence The ability to modify the logical schema without causing application programs to be rewritten. A database allows to build views over the schema s.t. different logical interpretations of the same data are possible. E.g. adding a new column to one user will not change the outcome for another user.

Declarative Language SQL is a declarative language and specifies how the result looks like while describing the tuples that should be part of it. A DL does not specify how to get to the result (i.e. no control flow, only logic of a computation - what, not how). With this, the DB can optimize queries.

Query Processing / Relational Algebra

Query Optimization Relational algebra allows to prove the equivalence of certain transformations. With this, a query can be optimized by the DB by first turning it into an operator tree (a plan) and then applying certain heuristics / additional information (all before the execution).

Transactions A ordered set of operations (reads and writes) which is only correct if all of them are completed = commit, else it is aborted. The state in between operations is inconsistent and consistent before and after (if correct).

ACID Principle See Figure 2.

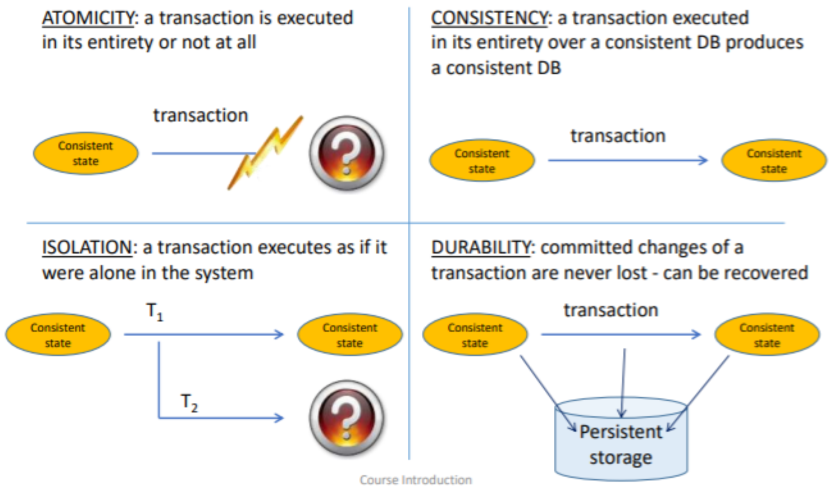

ATOMICITY: a transaction is executed in its entirety or not at all
Consistent state
transaction
CONSISTENCY: a transaction executed in its entirety over a consistent DB produces a consistent DB
Consistent state
transaction
Consistent state
ISOLATION: a transaction executes as if it were alone in the system
Consistent state
T_1
Consistent state
T_2
DURABILITY: committed changes of a transaction are never lost - can be recovered
Consistent state
transaction
Consistent state
Persistent storage
Course Introduction

Figure 2: The ACID principle.

Basic Database Architectures Shared nothing: each node runs its own data and its own engine, they share neither disk nor memory. Needs function shipping (query has to go where the data is) and / or data shipping (data has to go where query is). Easy to maintain and to scale, ideal when data can be partitioned / replicated and updates are not frequent. Shared memory: a coherent memory abstraction is provided over the network, merging the memory of each node, allowing for easier programming. Shared disk: data is stored in shared storage with a network typically attached to it, needs data shipping. Very common for cloud architectures. Separates storage from compute.

1.1 Reading Assignments

1.1.1 Architecture of a Database System

1.2 Exercises

1.2.1 Introductory Topics

2 Storage Management

2.1 Introduction

A big part of the performance of databases arises from proper storage management (movement through hierarchy), adequate data representations and suitable optimizations on organizing the data in memory.

Databases provide the illusion of large memory capacity and try to hide the performance problems created by implementing desirable properties (high bandwidth for seq. and concurr. access, low latencies for random accesses, persistent, etc.) through complex architectures and optimizations.

Two key guarantees we want to have provided by a DB storage system are: persistent and recoverable data (even in the event of a failure). We would also like physical data independence.

2.2 Memory Hierarchy

Databases go through the entire memory hierarchy frequently. Enhance locality (data organization, query scheduling), make sure data is available at layer where it is needed (pre-fetching), be clever about what to keep at each layer (caching, replacement) and keep track of modifications and write back to the lower layers (to persistent storage) when needed.

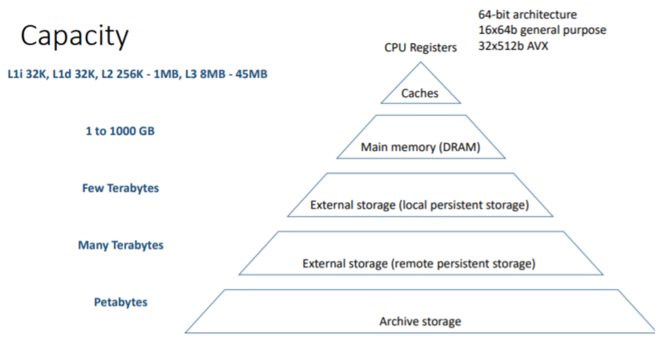


Figure 3: Memory hierarchy capacities.

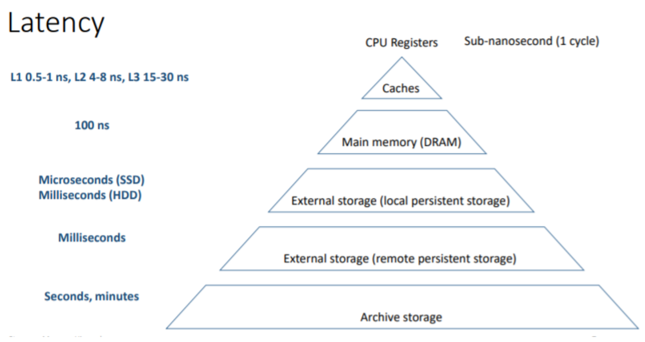


Figure 4: Memory hierarchy latencies.

Memory Wall Main memory suffers from several issues: there is never enough of it (application growth), memory outside the CPU chip (DRAM) is much slower than memory located inside of it (latency issue), processor-memory gap (processor speed increases faster than memory speeds - bandwidth is an issue), increased cost (DRAM is expensive) and main memory is not persistent.

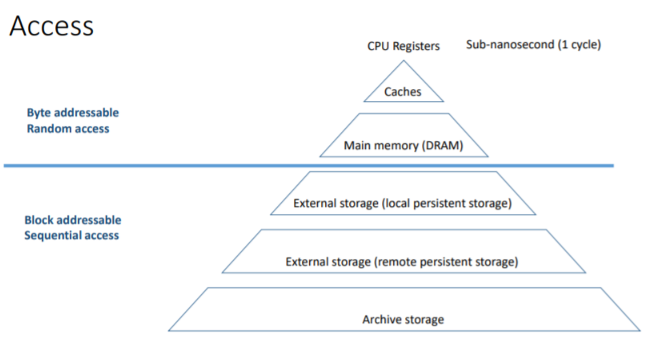


Figure 5: Memory hierarchy access methods.

Spatial Locality Put together what belongs together.

Temporal Locality Do at the same time things that require the same data.

Non-Volatile Memory (NVM) Located between DRAM and local external storage - combines characteristics of both. Byte addressable, random access, persistent, etc.

Cloud Computing Network between compute and storage layer. Ephemeral nature of computing infrastructure forces a separation of compute and storage. Flexibility is provided by cloud provider.

Network Attached Storage (NAS) Networks are becoming faster with more bandwidth than storage devices (e.g. RTT in a data center less than a seek operation on a HDD). Eventually, it might be faster to get data from the memory of a remote machine / storage device than from local disk. See also: Remote Direct Memory Access (RMDA).

Multicore and NUMA

Hardware Acceleration

SSD

2.3 Segments and File Storage

See the first layer in Figure 1. Assumptions: the standard database architecture is based on slow hard drive disks (HDD) that have a high latency for seek operations (random access) - most of the DB is **not** in memory.

Disclaimer: most explanations from the lecture are based on a real system (Oracle Database)[1].

Problem statement: a DB is doing many things at the same time and each thing (query, system process, DB component, etc.) active at any point in time needs its own logical view of the data. A DB creates such virtual, logical views of the system using its own mechanisms.

[1]For logical storage click here and here and for disk storage click here and here.

Logical and Physical Storage For Oracle 19, those two components are represented by an ER diagram seen in Figure 6 with a crow foot implying a one-to-many relationship).

A given logical object in a DB (table, index, etc.) is ”stored” in a tablespace, a tablespace is organized into segments, segments have space allocated to them in the form of extents and extents are sets of contiguously allocated data blocks.

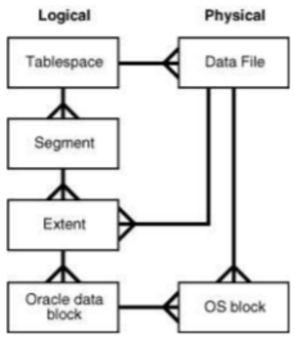


Figure 6: Oracle 19 logical vs. physical storage.

Tablespace Data unit in a DB which provides a logical representation of the principle of spatial locality (not necessarily stored continuously!) - some form of virtual memory. Data stored is either schema related (table, index, clustered tables) or engine related (data structures for the engine, e.g. result buffers). Space (memory / disk) is allocated to tablespaces.

Keep together what needs to be kept together = logical locality = one address.

Segment An object in the schema (or part of an object if it is partitioned). Consists of one or more extents (that can be in different data files). Segments allocate (virtual) space to objects. Usually when allocating space, a DBMS gives a little bit more to have some growing space.

Even if a table is partitioned into several segments across various disks, the table still has only one tablespace.

Extent A collection of physically continuous data blocks (size = tunable /dynamic parameter). Extents are mapped to one data file. Usually when allocating space for an extent, a DBMS gives a little bit more to have some growing space (1.25x common).

Extents are a compromise between static file mapping and dynamic block mapping. I.e. do static mapping to a set of blocks and if more space is needed dynamically do another static mapping connected to the first one. Needs an array of pointers = extent directory (which also needs space - want to keep it small).

Block The smallest amount / unit of physical space allocation. Kinda the same thing as a page in an OS - just bigger. The block size is a tunable parameter. With a small block size, you need to keep track of many physical locations and less actual sequential access. With a big block size, you will have fragmentation but more actual sequential access.

Block Structure: Slotted Pages A block has a header (address and type of segment, e.g. index, table, etc.), a table directory (schema of the table stored in the block), row directory (pointers to the actual tuples stored in the block (block, slot), data can be anywhere), free space and row data (tuples stored in the block). The directory grows downwards and the space for the tuples grows upwards.

No assumptions can be made on storage order of a table within an extent! E.g. if a tuple is deleted a new one can take its place that has been inserted much later.

Optimizing Block Usage

- **Percentage Free (PCTFREE):** Allow row inserts until x% of the space is occupied and leave the rest for updates to existing rows in the blocks. Avoids fragmentation occurring from updates that make an already existing tuple larger.
- **Percentage Used (PCTUSED):** Determines how much space needs to be free before free space is used to insert new tuples. No more new inserts until used space is under x%. Avoids having to constantly move a block from the used to the free list and vice versa (costly operation).
- These two concepts are often combined. They are a trade-off between efficiency and space.

Fragmentation Within Blocks Blocks can suffer from fragmentation. Compaction is expensive and only done when the block has enough space for an insert / update but the space is not contiguous. In case of an update not fitting the original block anymore, the tuple is put into a new block with a row pointer in the original space s.t. the ID of the tuple does not need to change (needs indirection) - too much of this slows down access since it multiplies I/O by a factor of at least 2.

Finding Space Segments contain one or more free lists that keep track (with pointers) of blocks that have usable free space (not done at an extent-level because the search for space would be more complex). Using several free lists helps avoid contention when performing parallel inserts / updates, e.g. different lists for different actions. Free lists are updates as transactions execute insert / delete / update statements using the rules established with Percentage Free and Percentage Used.

Writing to Disk A DB provides concurrency control (DB correct even if several transactions modified data at the same time) and recovery (data can be recovered even after a failure). More details in later sections.

Shadow Paging: When updating an attribute, copy it to a free slot (= page) and update it there. Once the transaction commits, mark the old slot as free and the new one as correct. Only makes sense when the overhead of creating a new copy and managing them is small enough (main memory, flash, NVM, etc.).

Delta Files: A copy of the attribute that is to be updated is stored in a delta file (instead of a free slot in the extent). The update is performed directly on the original value and the copy in the delta file can be discarded if it's no longer needed. Old data in delta file favors commits, simplifies undos and allows looking at older data. New data in delta file favors aborts and allows to delay the propagation of updates.[2]

[2] Oracle used rollback segments (segments used to store old copies of data) and now switched to undo-tablespaces.

2.4 Database Buffer Cache

See second layer in Figure 1. To process data, it has to reside in memory. Since not all the data fits into memory, databases cache blocks in memory and write them back to storage when dirty or in need of more space. The cache is not used for actual processing - intermediate results are stored in the database heap (region of memory where queries actually work on the data).

Similar to OS virtual memory / paging mechanisms but a DB has much better knowledge about access patterns and the process can be optimized to a larger extend.

It is important to keep the overhead of data structures in mind that are needed to keep track of things.

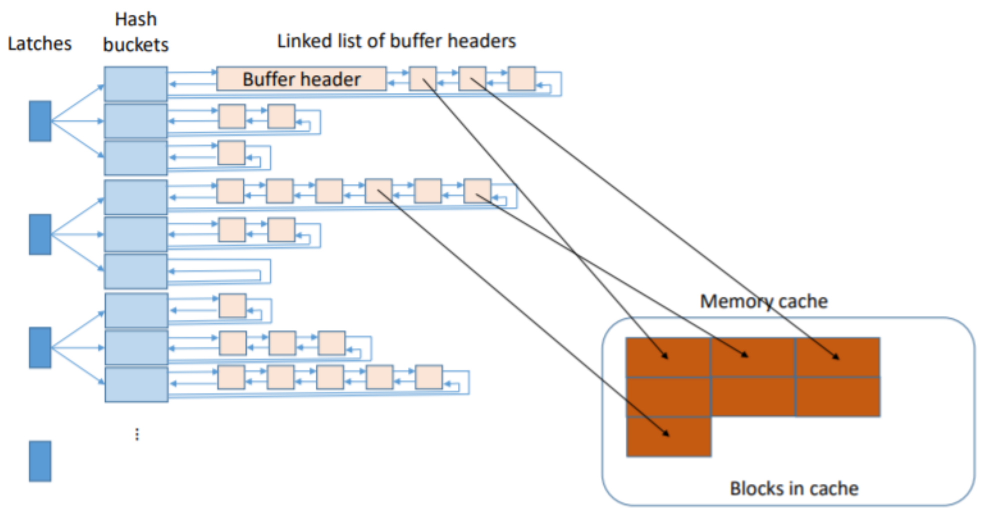


Figure 7: Architecture of a DB buffer cache.

Hash Table To check if a block is in memory, we take its ID and hash it. Look up the position indicated by the resulting value to check. This is the first check we need to do when processing data.

Latches Mechanism to access hash table(s) without the risk of a conflicting concurrent access (similar to a lock)[3]. A latch usually covers several hash buckets (tunable parameter). Latching per bucket or even per block header results in too much overhead since there are so many.

Performance Issues: To check if a block is in memory, a latch needs to be acquired. Since a latch can be only owned by one process, this process blocks several linked lists of block headers. Contention on these latches may cause performance problems (hot blocks, SQL statement accessing many blocks, similar statements executed concurrently, etc.).

Mitigation: Reduce amount of data in the block (with PCTFREE / PCTUSED), more latches and less buckets per latch, multiple buffer pools, tune queries to minimize number of blocks accessed (avoid table scans), avoid many concurrent queries / transactions accessing same data, etc.

Hash Bucket Correct linked list storing block header is found by hashing on some form of block identifier.

[3]Locks usually avoid conflicting updates to data by transactions, while latches avoid conflicting updates in system data structures.

Buffer Header Double Linked List After hashing, the linked list is traversed looking for an entry (block header) for the corresponding block. Keep the lists short since this is expensive (have more buckets). A block header contains lots of info (block number, type, format, LSN, integrity checksum, latches / status flags, buffer replacement info).

Some engines store other types of blocks in the linked lists, e.g. version blocks (updates result in copies with a timestamp inserted into the list, similar to shadow paging), undo / redo blocks for recovery, dirty blocks, pinned blocks, etc.

Block Status The following states are relevant for the management of the buffer: pinned (block cannot be evicted), usage count (statistics on specific block), clean / dirty. This info is used when implementing cache replacement policies.

Cache Replacement Policy What to cache, what to keep, how to evict and when, how to avoid thrashing cache with unnecessary traffic.

Least Recently Used (LRU): Keep track of when a block was used in a list - most recently used. When evicting, pick the least recently used block at the bottom. Does not work very well for DBs (in contrast to OSes) since large table / index range scans can pollute the cache without actually needing to re-access the cached data.

Modified LRU: Put rarely accessed data (statistics) at the bottom of the list or simply don't cache large tables. Or sort blocks according to access frequency instead of simple counters.

Optimizations

- Keep Buffer Pool: tell DB which blocks are important and should not be evicted (separate buffer).
- Recycle Buffer Pool: tell DB which blocks should not be kept after they are used (separate buffer).
- Keeping statistics of usage of tables and let system decide automatically what should / shouldn't be cached.
- Choose clean pages when needing to evict since it's faster (no write to storage).
- Ring Buffers: for scans, allocate the pages in a ring s.t. blocks are allocated only within the ring. Full buffer - evict pages from beginning of the ring as they have already been scanned.
- Since block sizes are not homogeneous, require a buffer cache for each block size for more space efficient replacement and simple management.

Touch Count (Hot / Cold List) A more sophisticated LRU. Insert new blocks in the middle of the list and keep a count of accesses. Frequently accessed blocks float to the top while rarely accessed ones sink. To avoid counting problems (many accesses only for a short amount of time), only increment counter after a (tunable) number of seconds. Periodically decrease counters.

Second Chance Strategies similar to LRU can become a bottleneck if the lists are large. With second chance, no list is maintained and counters are kept in the blocks. Buffer is treated as a circular buffer with an eviction process going around. When a page is accessed, counter = 1. When eviction process passes, if counter is 1, set to 0 - if 0, evict page.

Clock Sweep Same as second chance but takes into account that some pages are accessed frequently at regular intervals - uses a counter (with tunable max) instead of a 1/0 flag. Increase counter when touched, decrease when eviction passes, evict when counter = 0.

2Q: Using Two Lists Use a FIFO list for blocks that do not need to be kept and a LRU list for blocks that are accessed several times. If a block in the FIFO list is accessed again, it is moved to the LRU list. Blocks at the bottom of the LRU list are either moved to FIFO list (or immediately evicted). Evictions happen on FIFO list level.

2.5 Storage Techniques in Context

Instead of using physical storage, we are now looking at using a cloud storage and file system (such as Amazon S3). The previous material is put into context with a modern example of a database, namely *Snowflake*.

Snowflake A data warehouse specialized for analytical queries developed entirely on the cloud (cloud native). Separates compute (nodes running VMs with a local disk) from storage (Amazon S3).

The query processing layer (compute) is made up of virtual warehouses, with each consisting of a collection of worker nodes (EC2 instances in Amazon). Each worker node has a cache in its local disk with a simple LRU replacement policy. Metadata in the cloud services level tells us where things are stored.

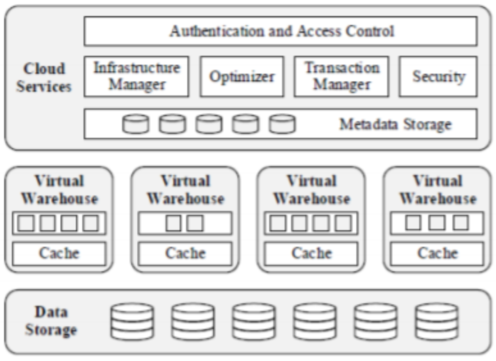


Figure 8: Snowflake architecture.

Amazon S3 (Simple Storage Service) Object storage service in the cloud that acts as he persistent storage that is available to applications. Unlike conventional local disks / distributed file systems (key-value where every object has a key, HTTP(S) PUT / GET / DELETE interface, no updates in place - only full writes that replace old object, can read parts of an object, high CPU overhead because of HTTP, extra expensive I/O because it's network based).

Micro-Partitions Snowflake's extents. To facilitate query processing, they are organized in a special way:

- Size ranges between 50 and 500 MB (before compression - data always compressed in S3). In contrast in other DBs, extents are typically in the order of KB.
- Each micro-partition has metadata describing what's inside.

- Metadata can be read without reading the whole micro-partition.
- Metadata is used to read just the part of the micro-partition that is relevant.
- Data in micro-partition is stored in columnar form (not by rows). This is the preferred storage format for analytics, improves cache locality, enables vectorized processing, facilitates projection operations and allows to process only the part of the table that is relevant.
- Horizontal partitioning (table partitioned into several MP horizontally) - allows to read the table in parallel and facilitates reading only wanted info.
- All of this allows for storage level processing to read only parts of the file that are needed which minimizes data movement to and from storage.

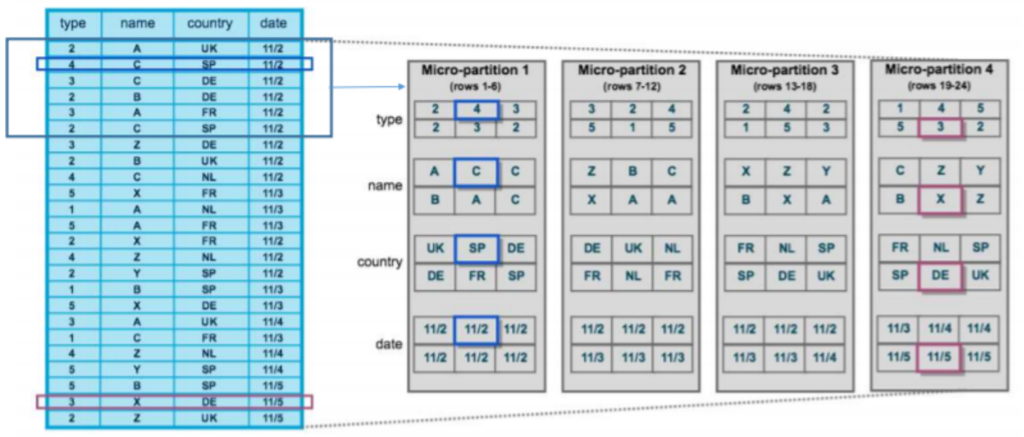

type
name
country
date
Micro-partition 1 (rows 1-6)
Micro-partition 2 (rows 7-12)
Micro-partition 3 (rows 13-18)
Micro-partition 4 (rows 19-24)

Figure 9: Logical and physical table structure in Snowflake.

Pruning based on Metadata Example of metadata: in this MP the min. age is x and the max age is y. With a query only wanting all entries with age > y, we can discard that MP. Other examples are: number of distinct values, number of nulls, bloom filters, etc.

Snowflake does not use indexes (requires a lot of space, induce random accesses which are bad for slow storage like S3, need to be maintained and selected correctly). Metadata is much smaller and easier to load.

Writing to Disk S3 does not support in-place updates (immutable), an object is replaced in its entirety. Snowflake uses this to implement snapshots of the data (shadow paging): write a new object when MP is modified and keep / discard old MP. Allows for reading of old data and provides fault-tolerance.

2.6 Reading Assignments

2.6.1 An Evaluation of Buffer Management Strategies for Relational Database Systems

2.6.2 Native Storage Extension for SAP HANA

2.6.3 The Snowflake Elastic Data Warehouse

2.7 Exercises

2.7.1 DBMIN

2.7.2 Column-Store

3 Access Methods

3.1 Introduction

We are now looking at layer 3 and 4 in Figure 1. We know now how data is brought from storage to memory - in this section we focus on how it is represented and organized in memory (how tuples are arranged within blocks and indexes), which impacts processing speed and hardware features that can be used.

Workload A DB system and its access methods can be constructed to handle a certain kind of workload (on-line transaction / analytic processing = OLT/AP). The type of workload also influences the choice of hardware. We have:

- **Transactional (OLTP):** Workload is dominated by (large amount of) short transactions with updates and modifications (high I/O), usually point queries (targeting one tuple), sensitive to contention, needs lots of processing power. Fits the traditional design of a relational engine.
- **Analytical (OLAP):** Workload is dominated by complex queries combining many tables, usually only read queries, needs large amount of memory and processing power. Usually uses a data warehouse system.

Trade-Offs Trade-offs made in a DB design can include:

- More compact representations vs. more complex processing.
- More indexes vs. more space and less maintenance cost.
- Row store for OLTP vs. column store for OLAP.
- The amount of unused space to keep for updates / movements.
- etc.

3.2 Pages and Blocks

Some of the information in this section was already discussed in previous sections.

In the previous section we have seen that data in a DB is stored in blocks, which are part of extents, which in turn are part of segments. There are different notions for a page:

- **Hardware Pages:** The atomic unit to write to storage, usually 4KB.
- **OS Page:** The unit used by the OS to implement virtual memory, usually 4KB.
- **DB Page:** Equivalent to a block, anywhere between 512B and 32KB. Trend is towards larger block sizes since they incur less overhead (less keeping track).

Finding a Block Each segment has a segment header which contains a (linked) list of used and free blocks. Both lists keep track of block IDs (Extent, Offset) while the free list also keeps track of the available space. This system is a potential bottleneck, especially for modification transactions.

Improving Performance: Using several free lists for faster concurrent transactions, making the traversal of the free list fast (sorting by size, keep it short, etc.), efficient search for holes in all blocks, etc.

Finding a Tuple / Record Each block, next to a block header, maintains a list of pointers (offsets) to all the slots in it that store a tuple each. Each tuple has an ID (Block ID, Offset). The slots / tuples are not uniform in size (hence the offsets).

Changes: Too large for the slot - keep a new pointer in original position to new location of tuple (can be in a different block, ID stays the same). Deletion - remove pointer. Smaller - just leave space empty. Insertion - use a block with enough free space (compact it if needed).

Tuple / Record Structure A tuple contains a header (validity flags for deletion, visibility info for concurrency control, bit map of null values, etc.) and attributes (data / pointer to data for each non-null attribute) as seen in Figure 10. A relational DB does **not** store schema information (table name, data types of attributes, number of attributes, etc.) in a tuple - this is in contrast to schema-less systems (see later).

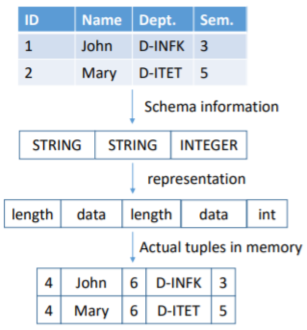


Figure 10: Tuple representation in memory (header omitted - usually in front of data).

Record Layout Optimizations The serial representation as seen in Figure 10 is intuitive but has a linear time to access each attribute. To improve the access time, we can either have a fixed sized part at the beginning of a tuple that stores offsets instead of lengths with each pointer pointing to the tail of an attribute (constant access time for each attribute or simply reorder the attributes such that variable length data (e.g. strings) is placed at the end (again with the pointer system from above).

Data Types

- **Integer Numbers:**
- **Real Numbers:** IEEE-754 standard for variable precision or fixed point representations for fixed precision (avoids rounding errors).
- **Strings and BLOBS (Binary Large Objects):** Length and data.
- **Time, Coordinates, Points, etc.:** System specific.

When a tuple / single attribute is very big (usually in reference to block size), instead of storing the whole thing one can store the fixed part of it and a pointer to the variable part (potentially in a different block or even a DB-external file). Since it is common that those attributes are not processed by queries, putting them somewhere else speeds up scanning operations. This is usually the case for BLOBS (e.g. pictures, text, etc.).

Row Store Tuples are stored as described so far with all the attributes staying together (just as seen as in a table). Allows for quick access and retrieval of an entire tuple. Is usually used in OLTP systems where operations are mostly carried out on individual tuples. When only retrieving a single attribute, a lot of unnecessary data is scanned and brought into the cache since we carry entire blocks.

Column Store The data is stored by columns, i.e. each block contains columns instead of rows of a table. Usually used on OLAP / in-memory DBs. Each column either has virtual IDs (order = ID, as in row store, safes space) or explicit IDs which are repeated for each column s.t. each column can be treated individually and (relative) order does not matter. Uses the cache very efficiently. Easy to compress. Improves bandwidth. Disadvantages: When an entire tuple is needed, we need to access several blocks. Complex if a tuple needs to be reconstructed as an intermediate / final result. Modifications are more difficult.

Vectorized Processing / SIMD: Simultaneously performing an operation on a vector of values. Column store is the perfect data representation for this. Very useful for numeric values and bit comparisons.

Partition Attributes Across (PAX) An alternative tuple representation as seen in Figure 11. A block is divided into mini-blocks and contains several tuples but is organized as a column store. Reconstructing the tuple does not require access to several blocks. See papers (below) for more info.

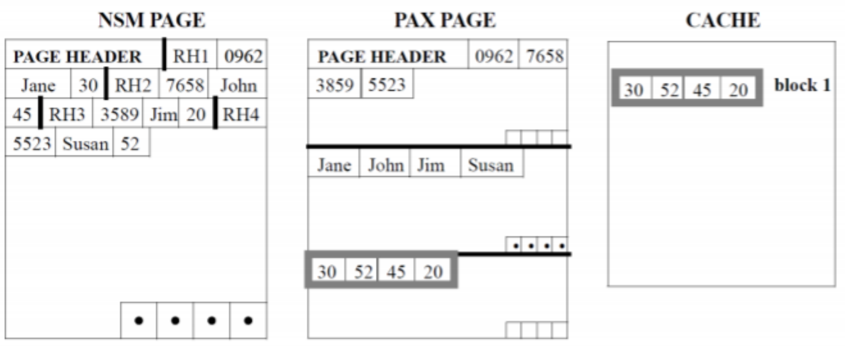


Figure 11: Row store (NSM) vs. PAX representation.

Compression Not used to save space but rather to save bandwidth. There is a trade-off between the CPU cycles needed to (de)compress and the memory bandwidth (today: former wins, CPU much faster than memory). There is a possibility to process data in its compressed form. There are many different compression mechanisms and they depend on the data organization (dictionary, run length encoding, delta encoding, bitmaps, etc.).

Dictionary Compression Rather an encoding than actual compression. Build a dictionary that maps long entries to, for example, integers / small numbers when the data is loaded. This can be applied to any finite collection of names (e.g. countries, departments, etc.). The data can be easily processed in its encoded form. A hash function is one way to implement a dictionary encoding.

Frame of Reference Many attributes have value locality and can be represented as a delta over some base (e.g. 1007, 1017, 1090 represented as 1000, 7, 17 and 90). This allows for operations on compressed data. Can be combined with delta encoding for sorted lists of data (store difference to previous value rather than actual value).

Run Length Encoding If a value repeats, store it once and how many time it appears. Useful for attributes with low cardinality (e.g. departments). Often used in column store. For row store it is used for long strings with repeated characters. Compresses the data significantly but processing is more complex. The encoding is variable in size.

Bit-Vector Representation / Bitmap For every value that an attribute may take, construct the bitmap as follows: create array as long as number of tuples, if tuple i has value x for that attribute, position i in the bitmap x is set to 1. Bitmaps act as an index and can be used to process queries just by looking at it (selections, joins on an attribute, group by on attribute, etc.). Can be further compressed using run length encoding - but processing becomes more complex.

3.3 Indexing

An index is a data structure that improves the speed (sub-linear time) of data retrieval operations (random and sequential) on a database table at the cost of additional writes and storage space (as in B+ Trees) - or processing power (as in hashing) - to maintain the index data structure.

An index can also be used to police database constraints when data is inserted / updated (i.e. unique, exclusion, primary / foreign key).

3.3.1 Hashing

Hashing can be used to construct an index but also as a partitioning strategy when allocating data.

Hash Function A function applied to some kind of arbitrary sized value (e.g. a single attribute value of a table) resulting in a fixed-size hash value. A hash value is usually much smaller than the original value. Different values might result in the same hash value since they are fixed in size (= collision).

There are many functions with strong properties but in a DB the function has to be computationally cheap (usually a modulo operation) since it is used very often.

Hash Table A structure that maps keys (usually primary key) to values. It uses a hash function to compute an index for an array of buckets / slots (in DB: bucket = typically a block) that contain the desired values. To look up a desired value, the key is hashed and the location of it is found with the resulting index. This only works for point queries.

Collisions do not happen with perfect hash functions but the table needs to be as big as the cardinality of the attributes (e.g. 4 byte keys = 4 GB table). But if the table is too small, we have too many collisions. Growing the hash table is an expensive operation. Choosing the perfect size is key.

Dealing with Collisions Since a bucket in a DB is usually a block and not a single tuple, i.e. colliding values might simply belong to the same block, we have less "bad" collisions - this usually means that we ran into a block that is full and we to allocate a new one. Ways to deal with such collisions include:

- **Chaining:** Add a block with free space to the linked list pointed to by a hash bucket. Not optimal because locating the desired tuple is now more complex (especially if the linked list is long). Improve efficiency by already adding free blocks to a linked list before collision even occurs.

- **Open Addressing:** Look for an empty slot in the hash table using some rule. Linear probing: simply add tuple to the block with index = index + 1. Cuckoo hashing: use several hash functions, if the first one leads to a collision use the next one and so on.

Growing the Hash Table This needs to be done when the hash table is completely full. A basic approach would be to create a new, larger hash table with more buckets (typically 2x) and rehash all the existing tuples. Very expensive and inefficient, lots of random accesses and cannot be done on disk (where hash indices are mostly used). Other approaches are often used in combination or in a nested manner (bucket directory pointing to another hash table pointing to actual data - helps with skew).

Extendible Hashing: Instead of having each hash value uniquely identify its own bucket, several hash values point to the same bucket and therefore the same block. If a block fills up, split it and move entries as needed. This increases the available space without significantly changing the hashing mechanism but it requires two page lookups (bucket directory and data block). To allow for more splitting (higher degree of sharing), the size of the table / bucket directory can be doubled by adding another bit to the matching value which increases the number of buckets = logical doubling.

Linear Hashing: A split pointer is used to indicate which bucket will be split in case of overflow (which is not necessarily the one that actually overflows - this one is simply chained) or other triggers (load factor, max. chain length, etc.). The entries targeted by the split bucket use a second hash function targeting the expanded range (mod n, mod 2n). After splitting, move the pointer to the next block. With this, the size of the hash table is gradually increased while redistributing the table (pointer hitting a chained block). The directory grows page by page instead of doubling. Once all buckets have been split, start anew.

3.3.2 B+ Trees

B+ Trees are used to create an index by trading in space. No matter what is said, a DB always uses B+ Trees, never simply B-Trees.

B-Tree Self-balancing tree that maintains sorted data and allows for searches, sequential access, insertions and deletions in logarithmic time. It requires linear space. It is a generalization of a binary tree since it allows for more than two children per node.

- Order = m.
- Every node has at most m children.
- Every non-leaf node except root has at least $\frac{m}{2}$ (round up) children.
- If it is not a leaf, the root has at least two children.
- A non-leaf node with k children contains $k-1$ keys.
- All leaves appear in the same level and carry no information.

B+ Tree Is a B-Tree where the data is at the leaves only (usually pointers to the tuples being indexed or more uncommonly the actual tuple). The leaf nodes are organized as a linked list. The values in the same leaf block might point to different data blocks.

Indices are referred by segments and they can use different block sizes (= nodes) than actual data blocks. Furthermore, an index can have its own memory buffer to avoid working on the index affecting working on data.

Clustered Index Forces the tuples of a table to be stored in the same order as the B+ Tree index indicates. With this, the table is physically stored in a sorted manner. An index might be defined over one or a combination of attributes (composite key). In the latter case, it is not possible to access the index with a subset of the attributes it is defined on. Typically done only for the primary key and automatic in systems that store the data directly in the leaf nodes. Most useful for tables that are not updated frequently.

A composite key is compared lexicographically:

$$(a1, b1) < (a2, b2) \iff (a1 < b1) \vee (a1 = b2 \wedge a2 < b2)$$

Non-Unique Values A B+ Tree index can be built on any attribute including those that are not unique. To differentiate duplicates, we can either repeat the key at the leaf nodes for every duplicate entry or store the key once and let it point to a linked list of all the matching entries. If the data is stored directly in the leaves, append the tuple ID to differentiate tuples the entries are referring to.

Direct Lookup To find a single tuple, traverse the tree starting from the root. Within each node, use binary search to look for the correct entry. At the leaf node: return the corresponding pointer / position.

Range Lookup To find tuples that are in a particular range, start by finding the leaf node corresponding to the first value and then follow the linked list until we hit the second value.

Insertion Lookup the corresponding leaf. If there is space, insert. If there is no space, split leaf into two, insert item and insert new separator on parent node. If parent node is full, split it in two and insert separator in parent's parent node. Go all the way to the root if necessary.

Deletion Lookup the corresponding leaf and remove entry. If the leaf is less than half order full, check a neighboring leaf. If it is more than half full, balance both. If it is half full, merge both. Update separator(s) in parent.

Concurrent Access Use lock coupling to protect index from conflicting concurrent accesses. When accessing the index, first lock the root and the first level. Check where you have to go and only then release first lock on root. Lock second level and repeat up to the leaf level.

Creating a B+ Tree Index A B+ Tree is created bottom up (leaves first):

- Sort the data
- Fill blocks one after another (leave space if updates are anticipated else we have a clustered and compact tree)
- Remember largest value in each block
- Create inner nodes by using largest value in each block as separator
- Iterate upwards to the next level until there is only a single block (root)

Optimizations

- **Reverse Index:** If the indexed attribute is a sequence and new items are constantly produced, insert items by reversing them so that they are more likely to go to different blocks. This protects from concurrent updates fighting to insert on the same block.
- **More Efficient Keys:** Replace keys in inner nodes with shorter separators that have the same effect or factor the children's common prefix.
- **Ignore Rules:** Don't merge nodes when they do not have enough data - delaying a merge can minimize changes. Periodically rebuild the tree instead. Also: using variable length keys might improve efficiency, but it is more complex.
- Wide / shallow trees are good for slow storage devices (large nodes, potentially over several sequential blocks) and deep trees are good for fast storage devices (small nodes, potentially several for one block).

3.3.3 Other Indexing Techniques

Query Selectivity Selectivity refers to the number of tuples returned by the query. A highly selective query returns very few tuples as a result (vs. low selectivity). The indices discussed above work well for queries with high selectivity.

For low selectivity, a table scan might be a good index.

Bitmaps Simple predicate selections can be very efficient using bitmaps by intersecting them for every queried column. Since bitmaps are usually sparse, they can be compressed pretty well (with run length encoding) and they are often used for large tables in OLAP - faster than other indices for low selectivity queries. Especially useful for data types where comparison is expensive.

Materialized Aggregates For groups of data that don't change very often, one can compute and store some basic statistics = aggregates (sum, avg, count, min, max, etc.). These can be used as small indices to check whether the data needed is within that group. They can also be used to compute aggregates over the whole table without having to read all of it.

Specialized Indices Text: tries, patricia trees, inverted indices (map words to documents). Spatial: r-trees, grid file. ETC

3.4 Access Methods in Context

Alternative designs for representing data in memory.

Denormalized Tables Usually, normal forms (rules) are used to eliminate redundancy in a DB schema by forcing table splits until each different table represents a distinct concept. This saves space but it is common that the split tables will be joined again in almost every query. Denormalized tables allow for more than one table to be stored in the same blocks by clustering the tables into the same segment (blocks are indexed by the common attribute) = materialized join. This reduces I/O and saves space but updates are more expensive. E.g. Clustered Tables in Oracle, see Figure 12.

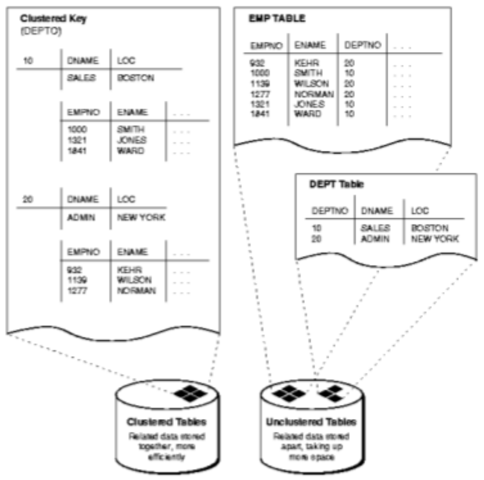


Figure 12: Oracle: Clustered Tables.

Log Structured Databases Instead of storing tuples and modifying them as needed, only store a record of how the data is modified (a log). With this, inserts record the entire tuple, deletes indicate that a tuple is invalidated and updates record modified tuples (no in-place updates). New log entries are simply appended at the end of the log file (much faster than random access). Mainly used for in-memory OLTP (minimizes cost for making data persistent) and cloud storage (file storage is typically append only). Not useful for heavy OLAP. To optimize this concept, the log file is periodically compacted (remove history by adding only one entry for each tuple - all operations are applied) and the tuples and their modifications are indexed.

No Indices Snowflake uses micro-partitions instead of indices (see previous section) and MonetDB (column store) uses database cracking: the index is built incrementally while the data is being processed on a column-wise basis. Initial queries are expensive but later cost us amortized as (some) work has already been done. See example in Figure 13.

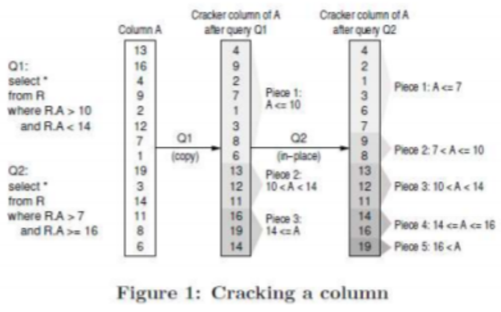


Figure 13: Database cracking example.

3.5 Reading Assignments

3.5.1 Modern B-Tree Techniques

3.5.2 The Design and Implementation of Modern Column-Oriented Database Systems

3.5.3 Data Page Layouts for Relational Databases on Deep Memory Hierarchies

3.5.4 A Hybrid Page Layout Integrating PAX and NSM

3.6 Exercises

3.6.1 Page Layouts and Indexing

3.6.2 Indexing

4 Query Processing

We are now at the top two layers of Figure 1.

4.1 Introduction

To process / parse a query sent from a client (DB application), it is first allocated to a process / thread through a DB engine interface while validating it, checking access control (permissions) and matching it with the query cache. Then, for each uncached query, a query plan (operator tree) is created (possibly rewriting and optimizing the query) and executed while buffers are read from the DB buffer cache or from disk. Finally, the results are sent back to the client.

A DB engine is commonly a shared environment with many users running queries concurrently while the engine itself runs its own maintenance procedures. A DB engine is similar to an OS since it has to schedule, orchestrate and mediate access to shared resources.

In a commercial setting, DBs are used programmatically meaning that queries generated by user interfaces are created by application programs. The use of views and query templates affects query processing.

Caching In a DB system, it is very likely to receive the same query multiple times and sometimes even in a short time frame. The best way to speed up execution is to reuse results from a previous execution (parsing, optimization, query itself, (intermediate) results[4], etc.) - everything can be cached! This also works well for data that does not change much (e.g. BLOBS / large data items). As always, consistency problems have to be dealt with.

Processes vs. Threads

- **OS Process:** A program execution unit scheduled and managed by the OS with a private and unique address space and its own state.
- **OS Thread:** A program execution unit part of a multi-threaded process that shares its address space and context with other threads from the same process. Scheduled and managed by the kernel.[5]
- **Client Process:** Execution unit running on client / application side used to connect to the DB engine.
- **Server Process:** Execution unit running inside the DB engine, used to execute queries on behalf of a client. A query can be processed by several server processes (parallel execution) a.k.a. a pool.
- Potentially confusing: the two things above can also be threads in a DB context!

With concurrent users, a DB system can be designed to either use a process or a thread for each user session. When using processes, we can take advantage of their isolation and security properties but scalability is limited, context switching is slow (lots of state is maintained) and implementation of parallel query processing is more complex. Furthermore, in a DB context, many data structures are shared across processes (buffer cache, lock table, common memory pools, etc.), which goes against the notion of processes. Most modern systems use threads (shared state and memory facilitates access

[4]Query results can be cached in the engine itself or outside of it (intermediate layer, client-side caching, etc.).

[5]Kernel: core program of an OS with complete control over everything in the system. It connects the application software to the hardware (CPU, memory, devices).

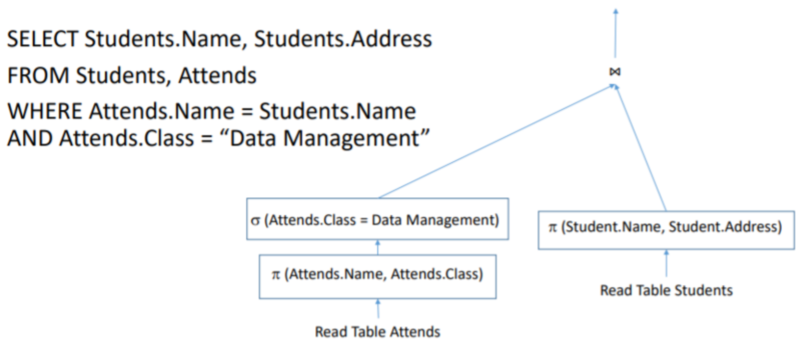
to common data structures and parallel query processing, lightweight, scalability, etc.) - but more complex management in terms of isolation.

Memory Structures Memory needs to be managed carefully since we have many different things running at the same time and shared vs. local memory has to be differentiated. It is common to divide it into regions each with a single purpose.

Cursor A pointer to a data structure (table, view, result, etc.) used to navigate the structure - similar to the iterator concept in programming languages. Space needs to be allocated to manage its state (opened, closed, accessed, etc.). When a table is processed, a cursor moves from tuple to tuple. Also used when delivering results to clients - instead of returning a possibly huge table, the cursor to it is returned.

4.2 Execution Models

When processing an SQL query, it is parsed into a tree of operators which is the basis for the execution model (see Figure 14). The leaves are the tables involved in the query and the root is the final result. Typically, each operator has at most two input from the lower layers (join). Operators can read input / output results one tuple at a time or as a whole - some are even blocking (all operations need to be completed until result can be issued). Each query and each subtree is its own table.



```
SELECT Students.Name, Students.Address
FROM Students, Attends
WHERE Attends.Name = Students.Name
AND Attends.Class = “Data Management”
```

⋈
σ (Attends.Class = Data Management)
π (Student.Name, Student.Address)
π (Attends.Name, Attends.Class)
Read Table Students
Read Table Attends

Figure 14: Query plan of an example SQL query.

Pull Mode An operator obtains data through a function call to a lower operator (control moves top-down) - data is obtained whenever it is needed. Good for disk-based systems and whenever the data doesn’t fit in memory.

Push Mode A lower operator sends its result up as soon as it completes processing a tuple / buffer / vector. Higher operators receive results and potentially have to buffer them if they’re not ready yet to process them. Can be more efficient than pull mode (hardware / CPU exploitation) but it’s more difficult to implement (buffer, synchronization, etc.) - similar to event-based programming.

4.2.1 Single Machine Execution Models

Iterator Model (or Volcano / Pipeline) Tuples = data traverse the tree from leaves to the root. Operators iterate over those tuples to process them by using the interface `Next()`. The execution (control) is top-down (the `Next()` call initiated at the top trickles down until it can be executed). Once both tables (left and then right) are filled, the nested for-loop is executed. Widely used in disk-based systems and works for all workloads. See Figure 15.

Pros: Generic interface for all operators (great information hiding). Iterators are easy to implement. Supports buffer management strategies. No overhead in terms of main memory. Supports pipelining, parallelism and distribution (with special iterators).

Cons: High overhead of method calls (context switches) and poor instruction cache locality (jumping down and up in the tree calling different functions).

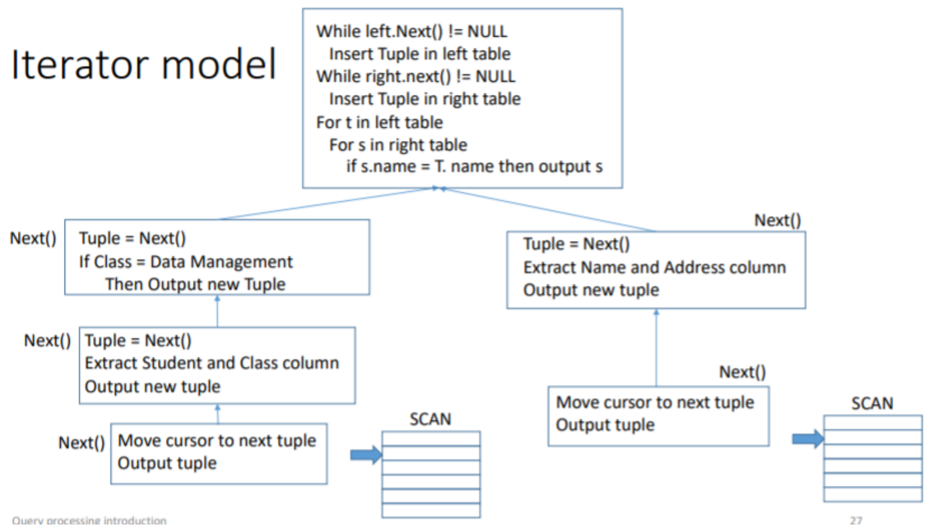

Iterator model
While left.Next() != NULL
Insert Tuple in left table
While right.next() != NULL
Insert Tuple in right table
For t in left table
For s in right table
if s.name = T. name then output s
Next()
Tuple = Next()
If Class = Data Management
Then Output new Tuple
Next()
Tuple = Next()
Extract Name and Address column
Output new tuple
Next()
Tuple = Next()
Extract Student and Class column
Output new tuple
Next()
Next()
Move cursor to next tuple
Output tuple
SCAN
Move cursor to next tuple
Output tuple
SCAN
Query processing introduction
27

Figure 15: Iterator model of example SQL query (typo: "Name" instead of "Student" in left block).

Materialization Model Similar to the iterator model but instead of outputting one tuple at a time, each operator outputs its result in a single buffer (each operator is called only once). Much less method calls - less overhead than previous model. Works well in OLTP (queries / transactions process small amounts of data resulting in small buffers that are passed around) - assuming data fits in memory. Not suitable for OLAP since data being passed around can be very large. See Figure 16.

Vectorized / Batch Model Exploits SIMD / AVX (and column-store) by combining the iterator and materialization model. Data is iterated over with `Next()` which returns a set of tuples (e.g. entire column) instead of a single one resp. a full buffer. This works best for OLAP systems.

4.2.2 Parallel Processing Execution Models

To parallelize a plan or distribute it across several machines, the above models can simply be extended with an `EXCHANGE` operator. The operator only moves data from one place (e.g. machine) to another - it does not modify data. See Figure 17.

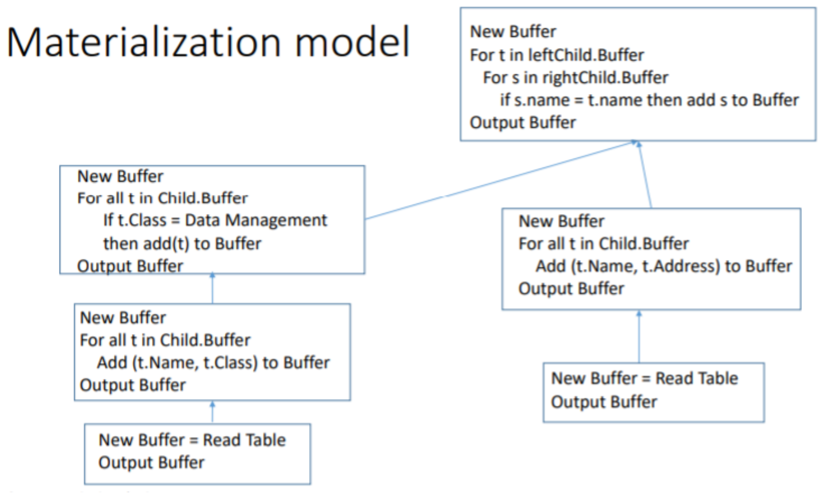

Materialization model
New Buffer
For t in leftChild.Buffer
For s in rightChild.Buffer
if s.name = t.name then add s to Buffer
Output Buffer
New Buffer
For all t in Child.Buffer
If t.Class = Data Management
then add(t) to Buffer
Output Buffer
New Buffer
For all t in Child.Buffer
Add (t.Name, t.Address) to Buffer
Output Buffer
New Buffer
For all t in Child.Buffer
Add (t.Name, t.Class) to Buffer
Output Buffer
New Buffer = Read Table
Output Buffer
New Buffer = Read Table
Output Buffer

Figure 16: Materialization model of example SQL query.

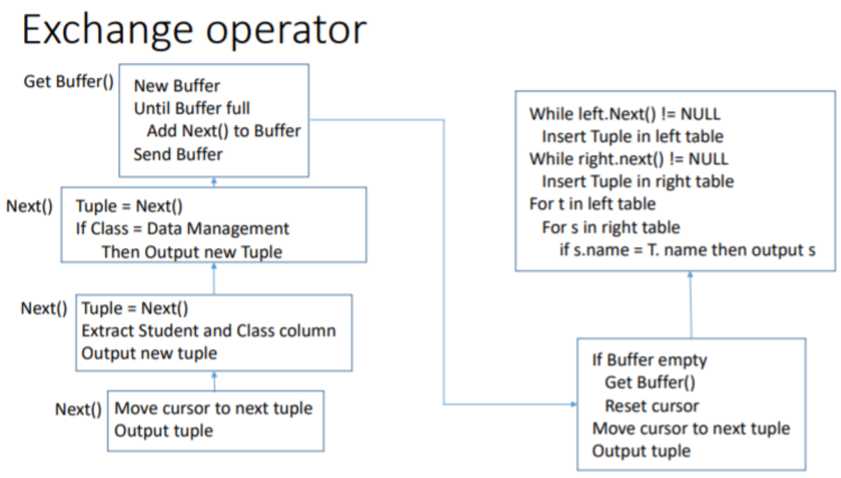


Figure 17: One way to implement an exchange operator (as a driver) shown with example SQL query.

4.3 Query Optimization

Since SQL is declarative, a DB engine has many options to translate a query into an executable program. After generating possible execution plans for a query, the best one has to be chosen (based on rules or based on estimated cost). A plan is influenced by the access methods for each table (leaves of the query tree), i.e. available indices, predicates, clustered tables (same extent), type of operator implementation, i.e. join implementation, sorted data, and the shape and form of the query tree.

View The result set of a stored query on the data (virtual table). A view can be queried just like any other persistent database collection object. Changes applied to the relevant underlying data are perpetuated. A view can me materialized into an actual table and added to the schema (usually done for common operations over the schema). Mainly used to implement logical data independence (create schema different from original one) and access control (access view instead of base tables).

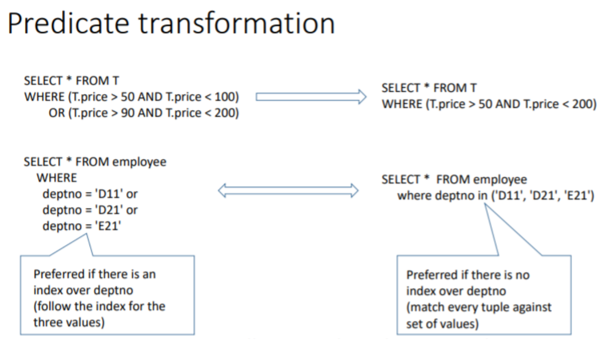
Schema A schema is made up of base tables defined at the beginning. It provides the basic organization of the data. Views allow to tailor that logical organization to the needs of particular applications without changing the basic schema. The more complicated a schema is, the more extensive is the use of views.

4.3.1 Query Rewriting

After a query has been parsed, it is often rewritten into an equivalent query based on the DB schema and on some heuristics. The rewritten query is then used as input for the query optimization process.

Rewriting can remove operations to make the query more efficient, can give the optimizer more freedom to operate, can make the query's intent more explicit and can map it to actual base tables and views as needed / for efficiency reasons.

Rewriting Predicates A predicate has to be checked for every tuple. The query will run faster if we either reduce the number of comparisons that need to be made and / or if we avoid going several times over the same tuple to check a different predicate. Predicates can be fully transformed or augmented (transitive closure) to decrease the number of comparisons. Augmentation gives the optimizer more options to consider (A=B, B=C - also add A=C; A=B, B>100 - also add A>100; etc.). Examples in Figures 18, 19 and 20.



Predicate transformation

```
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)
```

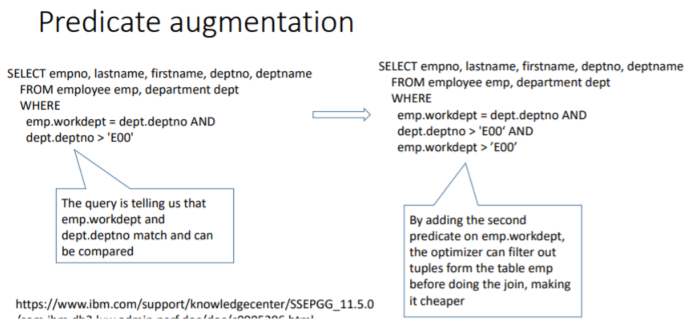
```
SELECT * FROM employee
  WHERE
    deptno = 'D11' or
    deptno = 'D21' or
    deptno = 'E21'
```

```
SELECT *  FROM employee
   where deptno in ('D11', 'D21', 'E21')
```

Preferred if there is an index over deptno (follow the index for the three values)

Preferred if there is no index over deptno (match every tuple against set of values)

Figure 18: Predicate transformation example.



Predicate augmentation

```
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'
```

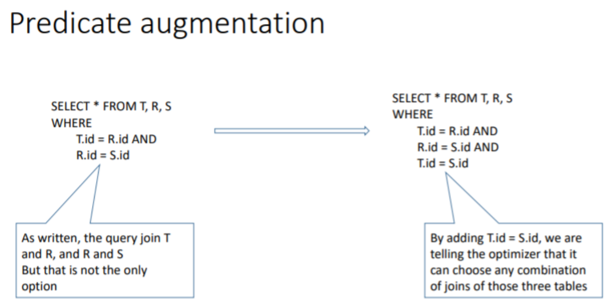
The query is telling us that emp.workdept and dept.deptno match and can be compared

By adding the second predicate on emp.workdept, the optimizer can filter out tuples form the table emp before doing the join, making it cheaper

https://www.ibm.com/support/knowledgecenter/SSEPGG_11.5.0

Figure 19: Predicate augmentation example 1.

- Predicate transformation
- Predicate augmentation
- Arithmetic changes (sum, count, avg)
- Predicate pushdown / view folding (view to base table for single pass)
- Query unnesting


Predicate augmentation

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

As written, the query join T and R, and R and S But that is not the only option

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

By adding T.id = S.id, we are telling the optimizer that it can choose any combination of joins of those three tables

Figure 20: Predicate augmentation example 2.

- Materialized views rather than base table
- Remove distinct if attribute is key
- etc.

4.3.2 Rule Based Optimization: Heuristics

With relational algebra, we can prove equivalence over queries (equivalence rules). This enables query transformations with the guarantee that the results stay the same.

- Conjunctive selection operations can be deconstructed into a sequence of individual selections.
- Selection operations are commutative thus can be applied in a different order.
- Selection of a cartesian product = theta join.
- Selection of a theta join = theta join with both predicates in conjunction.
- Theta / natural joins are commutative.
- Natural joins are associative.
- Theta joins are associative in a special manner:
-

4.3.3 Cost Based Optimization

To choose the best query plan out of multiple equivalent ones for a query, we might rely on its estimated cost based on many types of information.

Statistics Statistics that are constantly collected on tables, indices, buffers and the system itself can be an information source for query optimization (which plan and which operator implementation). We have:

- **Table Statistics:** Number of rows / blocks / etc., average row length, etc.
- **Column Statistics:** Number of distinct values / nulls / etc. (helps to estimate selectivity of a predicate or to decide on join order), data distribution (histogram), etc.
- **Extended Statistics:** Index statistics, number of leaf blocks, levels, clustering factor, etc.
- **System Statistics:** I/O performance and utilization, CPU performance and utilization, etc.

Histogram Help in cardinality estimation (CE), especially with skewed data. By default, uniform distribution of rows across the distinct values in a column is assumed

4.3.4 Operators

4.4 Reading Assignments

4.4.1 Sort vs. Hash Revisited

4.4.2 Query Optimization

4.4.3 The State of the Art in Distributed Query Processing

4.5 Exercises

4.5.1 Query Processing

4.5.2 Query Rewriting

4.5.3 Query Optimization

4.5.4 Cost-Based Optimizer

4.5.5 Query Operators

5 Transaction Management

5.1 Introduction

In the previous section, we focused on queries used to access the DB with the purpose of reading data (optimization, data movement, indices). In this section, we focus on transactions. Transactions are accesses to the DB with the purpose of modifying the data (update, insert, delete). Important is: correctness, recovery and state management.

Concurrency control and recovery are deeply interrelated. CC implementation affects recovery implementation and recovery implementation restricts what can be done in terms of CC.

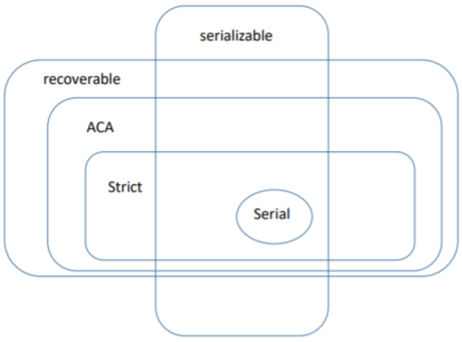


Figure 21: Consistency and recoverability properties.

5.2 Transaction Model

Transaction A unit of work made up of multiple operations (read and / or write) that has all the properties specified by ACID - operations inside the same transaction cannot be reordered. A transaction can either commit, i.e. its changes are made persistent, or abort / rollback, i.e. it is cancelled and all changes are reverted. Some systems include the concept of a savepoint, i.e. temporarily committing changes until a certain point with the possibility of rollback to a savepoint.

Transaction Manager Enforces concurrency control in a DB engine. It includes (see Figure 22):

- **Transaction Table:** List (commonly in common area) of active transactions in the system.
- **Transaction Handler:** Pointer to the structures containing all the relevant info related to a transaction (possibly in private area).
- **Lock Table:** Hash table containing entries that correspond to active locks. Locks on the same item = linked list.
- **Log:** Entries that capture the operations performed and are kept in memory until they're written back to disk.

A transaction manager goes through the following operations:

- **Begin T:** Create entry in transaction table (no log entry unless explicitly requested).
- **Read/Write:** Hash tuple ID to find corresponding entry in lock table. If empty: grant lock to T, else: attach request to list and grant request if compatible.
- **Write:** Create log entry (with LSN, before/after image, transaction ID, pointers to previous log entry of same T, etc.).

- **Commit T:** Release lock, potentially resume transactions waiting for lock, finalize log entries, write log entries to storage, potentially write modified data to storage, etc.
- **Abort T:** Release lock, potentially resume transactions waiting for lock, use log entries to undo/discard changes, potentially write log entries to storage, etc.

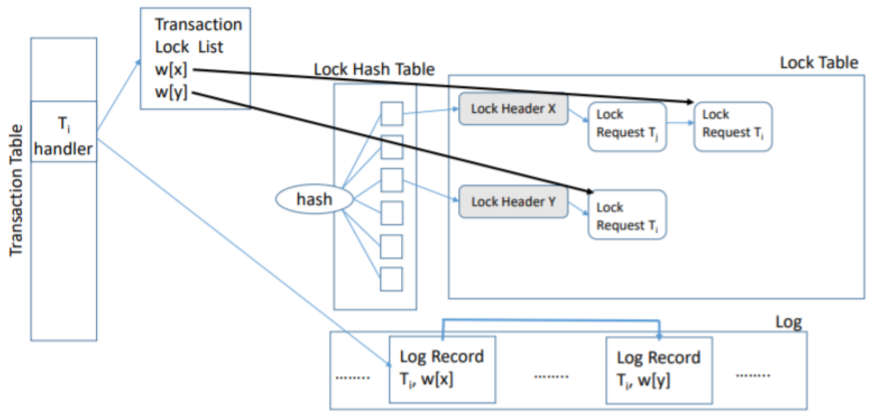


Figure 22: Basic functions of a transaction manager.

Conflicts

- **Write-Read (WR) / Dirty Read:** T2 reads object modified by T1 before T1 committed.
- **Read-Write (RW) / Unrepeatable Read:** T1 reads object that is modified and committed by T2, T1 then reads the same object again which results in a different value.
- **Write-Write (WW) / Lost Update:** T1 modifies an object previously modified by T2 with T1 committing before T2.

If no write is involved, read-read (RR) does not cause a conflict.

History / Schedule A partially ordered sequence of operations from a set of transactions.

5.3 Concurrency Control

How to ensure that the data remains consistent (data integrity) with a number of transactions and queries running at the same time.

ACID A set of properties of transactions intended to guarantee data validity despite errors, power failures and other mishaps. See Figure 2.

Atomicity: (Group of) operations taking place in their entirety or not at all. Intermediate states are not guaranteed to be correct. Atomicity requires isolation meaning that intermediate state should never be seen by another transaction.

Consistency: Assumption that an accepted transaction is correct (according to all defined rules / constraints).

Isolation: Property defining how / when the changes made by one operation become visible to others. Usually, transactions will be executed as if they were alone (intermediate state is not visible). Isolation is enforced with locking mechanisms.

Durability: Guarantee that transactions that have committed will survive permanently. Usually handled with some kind of logging mechanism, i.e. capturing consistent states in snapshots.

Serial History A history is serial if for every two transactions that appear in it either all operations of the first one appear before all operations of the second one or vice versa. By definition, a serial history with only committed transactions is correct (isolation, transactions start and end in a consistent state).

Equivalent Histories Two histories are equivalent iff they include the same transactions containing the same operations and conflicting operations of non-aborted transactions are ordered the same way in both histories. This implies that committed transactions in either history see the same state and leave the DB in the same state (same effects).

Conflict Equivalent Histories One can be transformed to the other by swapping non-conflicting operations.

Serializable History History is serializable iff it is equivalent to a serial history. If it is, it is correct (leaves the DB in a consistent state).

Serializability Graph A compact representation of the dependencies in a history. One node per committed transaction. Edge from T to T' if an action of T precedes and is in conflict with an action of T'.

Serializability Theorem A history is serializable iff its serializability graph is acyclic. Sorting it topologically results in an equivalent serial history.

Conflict Serializability If a history can be transformed into a serial history by swapping non-conflicting operations it is conflict serializable.

5.4 Recovery

How to ensure that the data remains consistent even when unexpected failures occur. A.k.a. changes from transactions that have not completed are not visible and changes from committed transactions are recorded and visible.

Types of Failures

- Transaction failure (abort, fail, time out, etc.) - undo its changes.
- System failure - bring DB back to a consistent state (undo and redo).
- Media failure (disk fail, permanent storage has errors, etc.) - restore DB to a known consistent state (replication, separating data from log files, etc.).

Recovery Procedures

- **R1:** Undo all changes from a single transaction - regardless of why.
- **R2:** Redo changes from committed transactions in case of a system crash where main memory is lost and disk is kept.
- **R3:** Undo the changes that remain in the system from active transactions in case of a system crash where main memory is lost and disk is kept.
- **R4:** Read consistent snapshots from backup and if possible apply the log (in case of system crash with loss of disk).

Undo vs. Redo To ensure atomicity in case of a non-completed transaction, we either need to: restore the initial DB state and remove any effects of the transaction (before image) or reach the intended final state of the transaction in question. Redo might also be applied to restore the changes of already committed transactions (after image).

Recoverable History (RC) If T2 reads an object modified by T1 and commits, T1 committed before T2 did. A committed transaction does not need to be undone since it cannot read wrong data and transactions commit in their serialization order.

Avoids Cascading Abort History (ACA) If T2 reads an object modified by T1, the read has happened after T2 committed. Aborting a transaction does not cause aborting others and transactions only read from committed transactions.

Strict History (ST) If T2 reads / writes an object modified by T1, the operation has happened after T2 committed or aborted. Undoing a transaction does not undo the changes of other transactions and transactions do not read / write updates of uncommitted transactions.

SQL Isolation Levels (ANSI)

Logging In case of system failure (sudden loss of data in memory) the engine needs to recover the DB up to the point of its last committed state which includes all changes made by all committed transactions up until time of failure = recovery procedure. The recovery procedure can only operate with data on permanent storage and needs to be correct even if successive failures occur in the middle of it. To do this, we need to be able to do logging.

Recovery Procedure A recovery procedure implemented by a recovery manager requires either undo and redo, just undo or just redo or neither. Undo needs before images (write back before commit) and redo needs after images (write back after commit).

Undo and Redo

- **Read:** Simply read value from block on buffer cache.
- **Write:** Create log entry (before and after image) and append to persistent log. Write after image to block on buffer cache.
- **Commit:** Write persistent log entry indicating that T has committed.
- **Abort:** For all updates, restore the before image using the log entry.

The recovery procedure will start from the end of the log and work backwards and keep two lists: undone items and redone items. Procedure terminates when all items are in either list or we're at the beginning of the log. For each log entry: if the accessed item is not yet in a list apply after image and add to redone list if it is part of a committed T or apply before image and add to undone list if it is part of an aborted T.

Just Undo

- **Read:** Simply read value from block on buffer cache.
- **Write:** Create log entry (before image) and append to persistent log. Write after image to block on buffer cache.
- **Commit:** Flush all dirty values modified by T if still in cache. Write persistent log entry indicating that T has committed.
- **Abort:** For all updates, restore the before image using the log entry.

Procedure same as above just with one undone list - each item part of an aborted T is restored to before image and added to this list. This relies on the fact that all committed values are in persistent storage and have not been lost. Works if strict execution is assumed.

Just Redo

- **Read:** If T has not written value before, read it from buffer cache. Else: read value from temporary buffer.
- **Write:** Create log entry (after image) and append to persistent log. Write after image to some temporary buffer.
- **Commit:** Apply all updates in temporary buffer to actual data blocks. Write persistent log entry indicating that T has committed.
- **Abort:** Discard temporary buffer.

Procedure same as above just with one redone list - each item not in list and part of a committed T is added to list and changes are redone using before image. Relies on never having dirty blocks in buffer cache - all data there is committed and is the most recent version.

Neither Rare resp. not used in practice, does not require a log and also there is no recovery procedure (that's the point). Uncommitted data is never written to persistent storage and data in memory is never dirty. Requires ability to write all changes made by a transaction to persistent storage in a single atomic action.

- **Read:** If T has not written value before, use current directory to find latest committed copy. Else: use shadow directory of T to find updated copy.
- **Write:** Write to buffer and add pointer to shadow directory of T.
- **Commit:** Create full directory by merging current one and shadow one of T. Swap pointer indicating latest committed directory.
- **Abort:** Discard buffer and shadow directory.

Log Record Log sequence number (LSN) for navigation, system change number (SCN) for event timestamps (T start), pointers to other log records of same T, transaction ID and related info (redo: change vectors = after images - describes changes to single block of data, undo: before images).

In Memory: In Storage:

5.5 Locking

Concurrency control in DBs is implemented using locking (usually with a lock table as seen in a transaction manager).

Compatibility Matrix

Two Phase Locking (2PL)

Strict 2PL

Deadlock Detection

Snapshot Isolation

5.6 Reading Assignments

5.6.1 Concurrency Control and Recovery in Database Systems

5.7 Exercises

5.7.1 Serializability and 2PL

5.7.2 Recovery