Advanced Database Systems

Author: Guanzhou (Jose) Hu 胡冠洲 @ UW-Madison CS764

Teacher: Prof. Xiangyao Yu

Query Processing & Buffer Management

Classical DBMS designs, SQL queries, analytical processing, buffer management, etc.

Join Processing

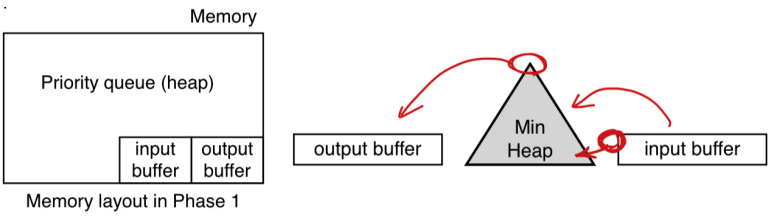
Link: https://cs-people.bu.edu/mathan/reading-groups/papers-classics/join.pdf

System architecture & assumptions:

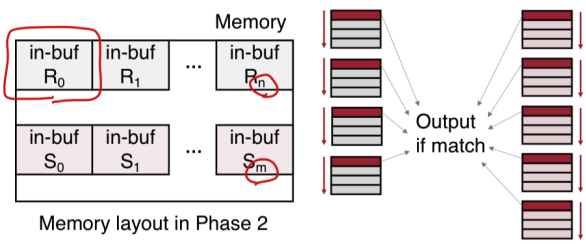
- Uniprocessor CPU
- 10s of MBs of main memory, good for both sequential and random accesses; denote memory capacity as $|M|$
- Big block-based disk, good for only sequential accesses
- Focus on only *equi-join* operation of two relations $R \bowtie S$, where $|R| \leq |S|$
- Use *fudge factor* F to account for any overhead amount

Four join algorithms discussed:

- *Sort-Merge join*
  - Procedure:
    1. Transform both relations into *sorted runs* of average size $2 * |M|$ each. 2 comes from using *replacement selection*: we first load $|M|$ blocks in memory and form a min-heap; instead of just dumping this min-heap as a $|M|$-sized run, we instead peek at the next tuple in input buffer -- if it $\geq$ all tuples that have been output, we can bring that tuple into heap to become part of the run -- in average $2 * |M|$ blocks per run

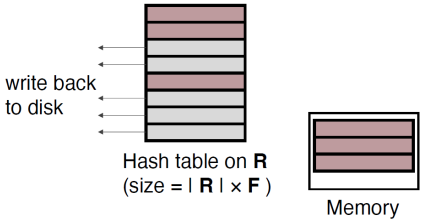


    2. This leads to roughly $\frac{|S|+|R|}{2|M|} \leq \frac{|S|}{|M|}$ runs in total. Do a *merge* by traversing through the runs and picking matching tuples



    3. If $|M| \geq \sqrt{|S|}$, then only two phases are needed because $|M| \geq \frac{|S|}{|M|}$ which means each run in phase 2 can get at least one slot in memory, meaning the merge can be done in one pass
- *Simple-Hash join*
  - Procedure:

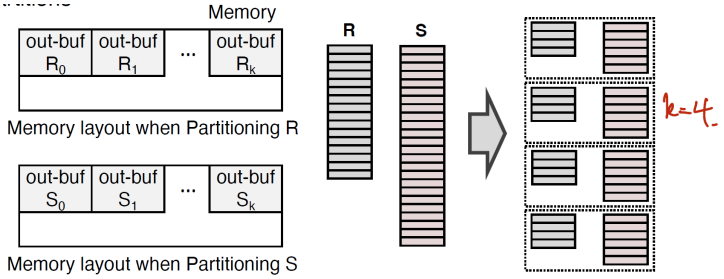
1. Build a hash table for R; if $|R| \cdot F > |M|$, then find a subset of buckets that fit in memory and build the hash table only containing those key ranges, and leave the rest of records for next pass



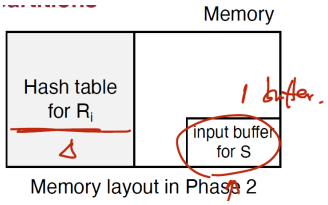
2. Traverse through S sequentially, and for each tuple, lookup the hash table to pick matching tuples
3. Repeat the above two steps until all tuples in R have been processed; R will get scanned in entirety for multiple times equal to the number of passes

- Good when $|R| \cdot F \leq |M|$ or is just slightly bigger, meaning the algorithm finishes in 1 or 2 passes and incurs no or little I/O traffic
- Otherwise, will incur significant I/O traffic by repeatedly scanning R and S multiple times

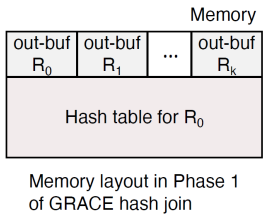
- *GRACE-Hash join*
  - Procedure:
    1. Partition both R and S into pairs of k shards using hash function bucketing, such that $|R_i| \cdot F < |M|$ for all partitions of R; requires $k \leq |M|$



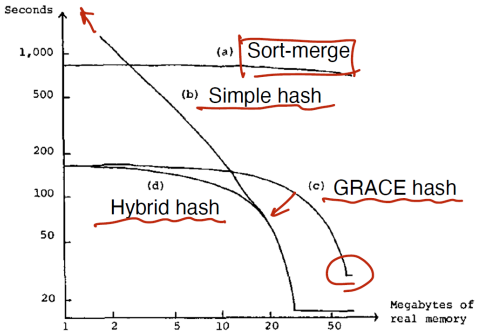
    2. Load in the first partition R_0 of R and build the hash table for it; requires roughly $\frac{|R|}{k} \cdot F \leq |M|$. Traverse through the partition S_0 of S and pick matching tuples



    3. Repeat step 2 for all k pairs of partitions
  - Good when $|R|$ is multiple times larger than $|M|$, because each table is effectively only written and read once
  - Not as good as simple-hash join when R fits in memory, which does not incur any I/O
  - Requires $|R| \leq \frac{|M|}{F}k \leq \frac{|M|^2}{F}$, i.e., $|M| \geq \sqrt{|R| \cdot F}$
    - If $|R| \cdot F > |M|^2$, phase 1 partitioning itself might require 2 or more passes
- *Hybrid-Hash join*
  - Combines simple-hash join and GRACE-hash join
  - Procedure:
    1. If $|R| \cdot F \leq |M|$, then do exactly simple-hash join; otherwise, do GRACE-hash join --
    2. During the phase 1 of GRACE-hash join, there will be memory capacity unused while partitioning R; use that to simultaneously build the hash table of R_0



    - If $|R| \cdot F = \alpha|M|$ where α is small, performance is close to simple-hash join, because R_0 is a significant fraction of R and is not written to disk
    - If $|R| \cdot F >> |M|$, performance is like GRACE-hash join



Other concerns & optimizations:

- Handling *partition overflow* when key space is skew and some buckets will be very large during hashing: just split that partition further into smaller key ranges
- Using Babb array (bitmap filter) to speed up lookups
- Applying *semi-join*: project R into only the join attributes, join to S, and then join the result back to R
  - useful if full R tuples won't fit in memory, but the join is selective and filter out many S tuples

Radix Join

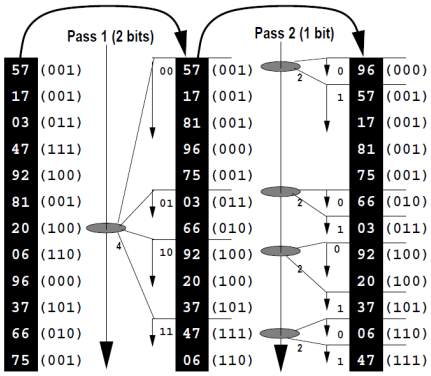
Link: https://www.vldb.org/conf/1999/P5.pdf

System architecture & assumptions:

- Machines hitting a memory wall, as memory capacity grows slower than CPU speed
- Want to optimize for for SRAM-cache/DRAM heirarchy
- Cannot simply apply partitioned hash join:
  - CPU cache has very limited capacity, yet TLB becomes a bottleneck if we have too many partitions
  - Memory layout of data partitions needs to be carefully design to avoid fragmentation

Radix join algorithm:

- *Radix partitioning* -- In each pass, cluster records based on the lower B bits of the integer hash value of key
  - For pass p, use next B_p bits starting from lest-most bit for partitioning
  - In order to know where in output array to write each record, need to scan the array twice -- first time collect size per partition and do *prefix-sum* to get the start index of each partition
  - Output array has identical structure as input array, meaning no complex memory allocation or fragmentation
  - Number of partitions per pass is determined by TLB size
  - Final size per partition is determined by CPU cache size



- Do radix partitioning for both relations, then join each pair of partitions using hash join (if partitions are relatively big) or nested-loop join (if partitions are all very small)
- Paper also touches on how non-integer join keys could be *dictionary-encoded* and stored in *columnar* format

DBMIN Buffer Management

Link: https://web.stanford.edu/class/cs345d-01/rl/DBBufMgmt.pdf

Classic *page replacement* policies:

- Least-recently-used (LRU)
- Most-recently-used (MRU)
- First-in-first-out (FIFO)
- Last-in-first-out (LIFO)
- CLOCK (second-chance FIFO)
- Random
- Many others... -- the optimal replacement policy *depends* on the data access pattern $\rightarrow$ access pattern in a DBMS is easier to predict than in e.g. OS or hardware, because of well-defined semantics of SQL queries

Query locality set model (QLSM):

- Key idea: decompose SQL queries into simple data access patterns
  - Each pattern has a heuristically optimal *locality set* size to be allocated as buffer space for it,
  - as well as a heuristically optimal replacement policy for it

- *Straight sequential* (SS): each page in a file accessed only once
  - e.g., `SELECT` on an unordered relation
  - locality set = 1 page
  - replacement policy = any
- *Clustered sequential* (CS): repeatedly read chunks, sequential inside each chunk
  - e.g., sort-merge join with duplicate join keys
  - locality set = size of the largest cluster
  - replacement policy = LRU or FIFO (buffer size $\geq$ cluster size), MRU (otherwise)
- *Looping sequential* (LS): repeatedly read entire file sequentially
  - e.g., nested-loop join
  - locality set = size of file being scanned
  - replacement policy = MRU
- *Independent random* (IR): truly random accesses
  - e.g., index scan through a *non-clustered* (e.g., secondary) index
  - locality set = 1 page or b unique pages in total
  - replacement policy = any
- *Clustered random* (CR): random accesses within each chunk
  - e.g., join between non-clustered, non-unique index as inner relation and clustered, non-unique outer relation
  - locality set = size of the largest cluster
  - replacement policy = LRU or FIFO (buffer size $\geq$ cluster size), MRU (otherwise)
- *Straight hierarchical* (SH): single traversal of index; similar to SS
- *Hierarchical with straight sequential* (H/SS): traversal of index followed by SS on leaves; similar to SS
- *Hierarchical with clustered sequential* (H/CS): traversal of index followed by CS on leaves; similar to CS
- *Looping hierarchical* (LH): repeatedly traversing an index
  - e.g., indexed nested-loop join
  - locality set = first few layers of the index
  - replacement policy = LIFO

DBMIN buffer management mechanism:

- For each `open()` file operation, estimate its QLSM
  - Allocate a set of buffer pages as its locality set
  - Choose a replacement policy for this set of pages
  - A query is allowed to run if its locality set fits in free frames
- If a frame contains a page not belonging to any locality set, it is managed by a global free list under LRU
  - If a page is requested and found in memory, but not in the file's locality set, detach it from the global free list and add into the locality set; this may lead to evicting some page determined by the local replacement policy back to the global free list
  - If a page is requested and not found in memory, it is first brought into the global free list by a disk I/O, then proceed as above
- To allow concurrent queries to share data pages, there's a global buffer table for queries to locate frames in other's locality sets

LeanStore

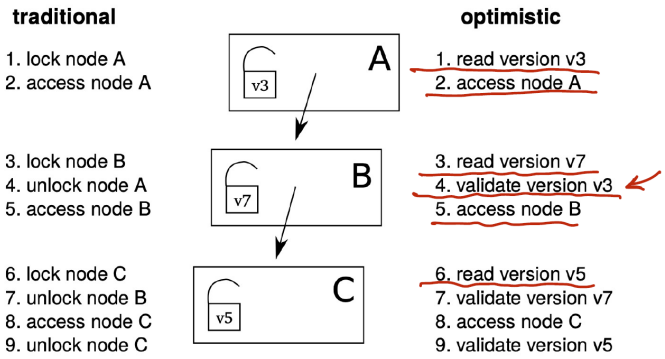
Link: https://db.in.tum.de/~leis/papers/leanstore.pdf

Modern main-memory database characteristics:

- The main memory contains (almost) full content of index structures and all data tuples; persistent storage is mostly for things like logs for durability concerns
- Data is managed at fine granularity, e.g., at tuple-level
- No indirection for buffer management: reference data through pointers

LeanStore optimizations:

- *Pointer swizzling*:
  - Pages residing in main memory are directly referenced using virtual memory address pointers
  - On-disk pages are still referred to by page ID
  - Each such pointer/ID is called a *swip*, with a bit indicating whether it's a pointer (swizzled) or page ID (unswizzled)
  - Each page has a single owning swip to prevent concurrency problems $\rightarrow$ in-memory data structures must be tree-like
  - Never unswizzle a page that has a swizzled children $\rightarrow$ when trying to unswizzle a page, loop through all its children pointers -- if there is a child that is swizzled, unswizzle the child instead
- *Page cooling*:
  - Randomly add pages to cooling stage, unswizzle them but not replacing them
  - Colling pages enter a FIFO queue, and a page is replaced it if reaches the end of queue
  - Upon access, a cooling page is swizzled
  - Hot pages' access tracking information do not need to be updated upon every access
- *Optimistic latching & lock coupling*:



- *Epoch-based reclamation*: evict a page from cooling stage only if all threads have finished reading it

Access Path Selection

Link: https://courses.cs.duke.edu/compsci516/cps216/spring03/papers/selinger-etal-1979.pdf

IBM System R query optimization:

- Cost of an *query execution plan* ≈ I/O cost + computation cost $= \#I/Os + W \cdot RSICARD$
- Goal: enumerate execution plans and pick the one with the lowest cost
- Statistics notation:
  - $NCARD(T) =$ #tuples in table T
  - $TCARD(T) =$ #pages in table T
  - $P(T) =$ fraction of pages in segment that belongs to T, can assume 1 if segment belongs to T entirely
  - $ICARD(T) =$ #distinct keys in index I
  - $NINDEX(I) =$ #pages in index I
- A *segment* contains disk pages that can hold tuples from multiple relations
  - A *segment scan* is basically sequential scan of all pages in segment
  - An (clustered or non-clustered) *index scan* utilizes an index to find high-key and low-key positions
- *Sargable predicates* (search-arguments-able predicates) are those that can be passed into RSS and have filtering applied early
  - Each clause is put in conjunctive normal form (CNF); each term is called a *boolean factor*
    - a boolean factor := `attribute COMPARISON_OP value`
  - These are non-sargable examples:
    - `function(attr) = something`
    - `attr1 + attr2 = something`
    - `attr + val = something`
    - `attr1 > attr2`
  - A predicate *matches* an index iff.
    - the predicate is sargable, AND
    - columns references in the predicate match an *initial subset* of attributes of indexing keys sequence

Estimation of computation cost -- estimating the *selectivity factor* F of predicate CNF to get $RSICARD$:

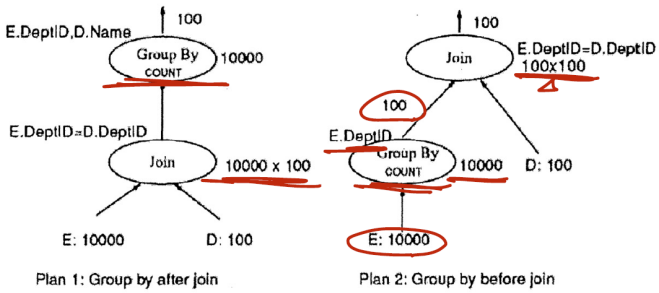
- `attr = val`: if index exists, $F = \frac{1}{ICARD(I)}$; else, $\frac{1}{10}$
- `attr1 = attr2`: $F = \frac{1}{\max\{ICARD(I_1), ICARD(I_2)\}}$
- `attr > val`: $F = \frac{val_{high} - val}{val_{high} - val_{low}}$
- `pred1 AND pred2`: $F = F_1 \cdot F_2$
- `pred1 OR pred2`: $F = F_1 + F_2 - F_1 \cdot F_2$
- `NOT pred`: $F = 1 - F_p$

Estimation of I/O cost -- estimating #page accesses:

- Segment scan: $IO = \frac{TCARD(T)}{P(T)}$
- Unique index point matching: $IO =$ 1 data page + several index pages
- Clustered index predicate matching: $IO = F \cdot (NINDEX(I) + TCARD(T))$
- Non clustered index predicate matching: $IO = F \cdot (NINDEX(I) + NCARD(T))$

Access path enumeration for a query:

- A tuple order is an *interesting* order if it is one that is specified by the query block's `GROUP BY` or `ORDER BY` clauses
  - Each sorted index can give us some tuple order
  - To find the cheapest access plan for a single relation query, we examine the access paths that produce tuples in each interesting order, as well as unordered access paths + the cost of sorting the output result
- Join ordering: right child is the inner relation
  - *Left-deep tree*: for nested-loop join or hash join, left-deep tree allows pipelineing
  - *Right-deep tree*
  - *Bushy tree*: may produce cheaper plans, but are rarely considered due to explosion of search space
- `GROUP BY` push-down could help reduce join cost:



C-Store

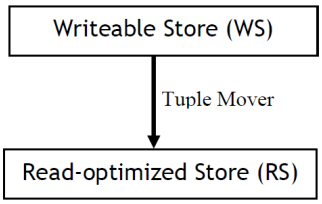
Link: https://web.stanford.edu/class/cs345d-01/rl/cstore.pdf

Row store vs. Column store:

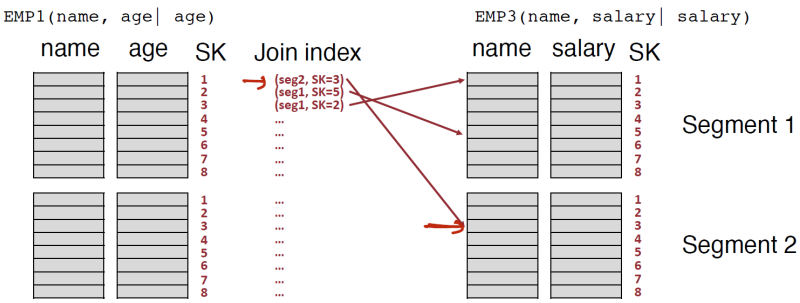
- Row format is write optimized and suits transaction processing (OLTP)
- Column format is read optimized and suits analytical processing (OLAP); easier to compact and compress data; less efficient to update records

C-Store design:

- Shared-nothing architecture with possible replication for redundancy; each node runs two stores, with writes go into the WS and the columnar RS, with periodic migration



- Data model:
  - A *projection* is a group of columns sorted on the same attribute(s); An attribute can belong to multiple projections and be sorted in different orders
  - Each projection is horizontally partitioned into *segments*, also called *row groups*
  - Each tuple in a projection is associated with a *storage key* (SK) denoting the logical row it belongs to
    - In RS, SKs are not explicitly stored and are simply the physical index of record in segment
    - In WS, SKs need to be explicitly stored
  - *Join indices* store the mapping between projections that are anchored at the same table with one-to-one mapping



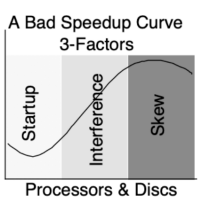
- Data encoding choices:
  - *Run-length encoding* (RLE) for self-ordered, few distinct values
  - *Bitmap encoding* for foreign references, few distinct values
  - *Delta encoding* for self-ordered, many distinct values
  - No encoding for foreign references, many distinct values

Parallel DBMS

Link: https://people.eecs.berkeley.edu/~brewer/cs262/5-dewittgray92.pdf

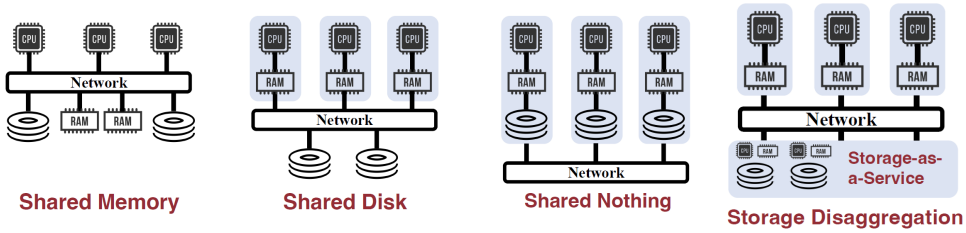
Parallel DBMS metrics:

- *Speedup* = small system elapsed time / big system elapsed time; linear $\rightarrow N$



- *Scaleup* = small system elapsed time on small problem / big system elapsed time on big problem; linear $\rightarrow 1$

Design spectrum:



Given a query plan, ways to make it parallel:

- *Pipelined parallelism*
- *Partitioned parallelism*
  - Round-robin
  - Range partitioning
  - Hash partitioning
  - See slide for pros & cons of each: the same old stuff mentioned many times in many courses...
- Parallelism *within* each relational operator
  - Each operator has a set of input and output ports
  - Connected through merge operators to keep SQL operators unmodified
  - Requires *data shuffling*

Transaction Processing

Transactions, ACID, locking & concurrency control, durability & recovery, etc.

Granularity of Locks

Link: https://web.stanford.edu/class/cs245/readings/granularity-of-locks.pdf

ACID properties of transactions concurrency control:

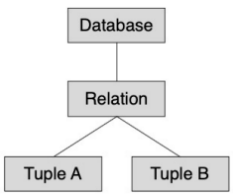
- **A**tomicity: transaction happens all-or-none
- **C**onsistency: app-layer integrity constraints are satisfied (NOT the consistency property in distributed replication)
- **I**solation: how operations from concurrent transactions are allowed to interleave
- **D**urability: committed transaction's effects must persist across system failures

Problems with traditional locking:

- Traditionally, locks only have two modes: *exclusive* (X) on writes and *shared* (S) for reads
- Consider a database with many tables, each consisting of many tuples, and consider multiple concurrent clients -- the traditional locking mechanism puts a dilemma on the *granularity* of locks:
  - Huge DB lock: simple & efficient, but locks everyone out; good if I'm doing a really bulk operation, bad if I'm only interested in a small tuple
  - One lock per tuple: finer-grained, but can easily have too many locks, leading to poor performance

Hierarchical Locking:

- Organize resources as a tree (or more generally, a DAG), for example:



- Version #1: still only have X and S modes; locking an internal node on X or S *implicitly* locks all its descendant nodes with the same mode
  - If I want to scan a file, better hold one S on the file node
  - If I want to write a few tuples, better hold Xs on the individual tuples
  - S and X on the same node are not *compatible* -- attempting to acquire S on a node already having X will fail, and vice versa
  - Problem: What about conflicting transactions, e.g., one trying to take S on a file and the other trying to take Xs on some tuples within it? This should not be allowed.
- Version #2: introduce *intention* modes on internal nodes: *intention exclusive* (IX) and *intention shared* (IS)
  - To lock a node with X mode, must traverse the tree from root and lock all ancestor nodes along the path with IX
  - To lock a node with S mode, must traverse the tree and similarly lock all ancestors with IS
  - S and IS are compatible -- attempting to acquire S on a node already having IS is ok, and they will share reading permissions of some children
  - X and IX are not compatible -- attempting to acquire X on a node already having IX is not ok, because some children must already be locked in X
  - IS and IX are compatible -- attempting to acquire IS on a node might just mean attempting to acquire S on a disjoint set of children from those locked in X; conflicts, if any, will be resolved at deeper level
  - Problem: Consider a workload that scan a big table while only attempting to update a few tuples along the way. With the current version, it must either hold a big X lock on the table, or hold many S locks on read-only tuples, which is not ideal.
- Version #3: introduce a combined optimization mode: *shared and intention exclusive* (SIX)
  - For the abovementioned workload, can hold a SIX lock on the table and a few X locks on the tuples to modify
  - SIX and IS are compatible -- can have disjoint sets of children locked in X and S, respectively, by the two transactions; conflicts, if any, will be resolved at deeper level
  - SIX is not compatible with all other modes -- similar reasoning follows

	NL	IS	IX	S	SIX	X
NL	YES	YES	YES	YES	YES	YES
IS	YES	YES	YES	YES	YES	NO
IX	YES	YES	YES	NO	NO	NO
S	YES	YES	NO	YES	NO	NO
SIX	YES	YES	NO	NO	NO	NO
X	YES	NO	NO	NO	NO	NO

- Lock are requested from *root to leaf*
  - and released from *leaf to root*, OR
  - released at the end of the transaction as an atomic operation

Other issues & extensions to locking:

- *Semantic locking*: you can have more lock purposes than reads and writes; e.g., increments can have its own semantic and be compatible with other concurrent increments

	S	INC	X
S	Y	N	N
INC	N	Y	N
X	N	N	N

- Lock granting scheduling may be altered to improve parallelism
  - however, need to avoid *starvation* (delaying a transaction indefinitely)
- Deadlock solutions:
  - *Deadlock detection*: maintain a dependency graph (a "wait-for" graph)
  - *Deadlock prevention*:
    - *No-Wait* -- whenever encountering a conflict, abort
    - *Wait-Die* -- a transaction waits if it has higher priority than those in queue, otherwise abort
    - *Wound-Wait* -- a transaction preemptively aborts queued ones if it has higher priority, otherwise wait
  - *Lock ordering*: everyone acquiring locks in a pre-defined order, e.g., by global memory address

Isolation Levels

Link: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf

Definition of ANSI isolation levels:

- *Serializable* (SR or SER) :$=$ as if transactions are executed in a serial order
  - $\equiv$ long read locks + long write locks
- *Repeatable read* (RR) :$=$ serializability but allowing *phantom effects*
  - Range scans under a predicate may not see newly inserted entries committed by others
  - $\equiv$ long record read locks + short predicate read locks + long write locks
- *Read committed* (RC) :$=$ repeatable read but allowing *fuzzy read*
  - Multiple reads of the same record within a transaction may get different values
  - $\equiv$ short read locks + long write locks
- *Read uncommitted* (RU) :$=$ read committed but allowing *dirty read*
  - Read may see dirty data of other uncommitted transactions' writes
  - $\equiv$ no locking on reads + long write locks
- No isolation :$=$ read uncommitted but allowing *dirty write*
  - Pretty useless isolation level -- no practical DBMS will provide this level
  - $\equiv$ no locking on reads + short write locks (or no locking on writes)

Other useful isolation levels:

- *Cursor stability* (CS) :$=$ using a cursor implementation (pointer to some record during some range scan), the value of record currently pointed to by the cursor cannot be modified during the period when cursor does not move
  - RC $<<$ CS $<<$ RR
  - Prevents the *lost update* phenomena from RC
  - $\equiv$ read locks held on current of cursor + short predicate read locks + long write locks
- *Snapshot isolation* (SI) :$=$ transaction reads records *versioned* at its *start timestamp* and writes records at its *commit timestamp*
  - RC $<<$ SI, SI $<>$ RR, SI $<>$ SER
  - Allows the *write skew* phenomena from SER: transactions can see a snapshot that does not reflect the latest updates and thus end result is not serializable
  - SI requires read snapshot to reflect all the changes that committed before transaction starts (i,e,, *real-time* constraint), which is not necessarily enforced by SER
    - Serializability + real-time constraint == *strict serializability* (i.e., *transactional linearizability*)
    - If T_1 commits before T_2 starts, T_1 must precede T_2 in the serial order
  - Can be implemented naturally in a *multi-version concurrency control* (MVCC) database

Two-phase locking (2PL) is the canonical way of ensuring desired isolation:

- Definition:
  - Growing phase -- acquiring locks, no releasing
  - Shrinking phase -- releasing locks, no acquiring
  - *Serialization point* is when after all locks are acquired but before any release
- *Strict 2PL* == 2PL + all `x` locks released *after* transaction commits
  - i.e., transactions only hold *long locks* (release all at once at the end of transaction) $\Rightarrow$ ensures *serializability*

- Alternatively, using vanilla 2PL, could trigger *cascading aborts* of all transactions that observed dirty writes; this requires tracking effort and also does not defend against fuzzy reads, etc.

Optimistic Concurrency Control (OCC)

Link: https://www.eecs.harvard.edu/~htk/publication/1981-tods-kung-robinson.pdf

Different concurrency control schemes:

- *Pessimistic concurrency control* (locking) resolves conflicts eagerly
  - Locking overhead could be high, even for read-only transactions
  - More vulnerable to deadlocks
  - Limited concurrency due to congestion and holding long locks
- *Optimistic concurrency control* (OCC) ignores conflicts during execution and resolve conflicts (through *validation*) at commit time
  - OCC may lead to high abort rate under high concurrency, since conflicts do not always mean non-serializable history, but OCC will abort the transaction to let it retry (while if using locking, just some other writing transaction might be slightly blocked and delayed)
  - OCC has no native support for *prioritizing* transactions to ensure their commit
    - Polaris protocol
    - May also fall back to 2PL if a transaction has aborted for too many times
- *Timestamp ordering* (T/O) does *time-traveling* to try to put a committing transaction at an appropriate timepoint
- *Multi-version concurrency control* (MVCC) supports snapshot isolation

Classic OCC algorithm:

- 3 phases: Execution - Validation - Commit
- *Execution* phase (originally named *read phase*) buffers writes locally; operations:
  - `tcreate`: create a new record

$$
\begin{array}{l}
\textit{tcreate} = (\\
\quad n := \textit{create};\\
\quad \textit{create set} := \textit{create set} \cup \{n\};\\
\quad \textbf{return } n)
\end{array}
$$

  - `twrite`: write to local *write set*, no visible modification to global database yet

$$
\begin{array}{l}
\textit{twrite}(n, i, v) = (\\
\quad \textbf{if } n \in \textit{create set}\\
\qquad \textbf{then } \textit{write}(n, i, v)\\
\quad \textbf{else if } n \in \textit{write set}\\
\qquad \textbf{then } \textit{write}(\textit{copies}[n], i, v)\\
\quad \textbf{else } (\\
\qquad m := \textit{copy}(n);\\
\qquad \textit{copies}[n] := m;\\
\qquad \textit{write set} := \textit{write set} \cup \{n\};\\
\qquad \textit{write}(\textit{copies}[n], i, v)))
\end{array}
$$

  - `tread`: read from either local write set or global database; maintain *read set*

$$
\begin{array}{l}
\textit{tread}(n, i) = (\\
\quad \textit{read set} := \textit{read set} \cup \{n\};\\
\quad \textbf{if } n \in \textit{write set}\\
\qquad \textbf{then return } \textit{read}(\textit{copies}[n], i)
\end{array}
$$

  - `tdelete`: mark deletion in local *delete set*, no deletion from global database yet

$$
\begin{array}{l}
\textit{tdelete}(n) = (\\
\quad \textit{delete set} := \textit{delete set} \cup \{n\}).
\end{array}
$$

- *Commit* phase (originally named *write phase*) makes all modifications global, reflects them to global state
- *Validation* phase determines whether the transaction can *commit* or must *abort*, by checking if there are concurrent transactions that have conflicts with me that will lead to violation of isolation level (assuming serializability here)
  - Transaction is assigned a *transaction number* tn when it enters the (critical section of) validation phase
    - This number, which is $finish\ tn$ as presented below, determines the global serialization order
    - Atomically fetches from (and increments) a global tnc counter
  - Serial validation:
    - At transaction start, grab a $start\ tn$
    - At validation, grab a $finish\ tn$ and check for set intersections

$$
\begin{array}{l}
\textit{tend} = (\\
\quad \langle \textit{finish } tn := tnc;\\
\quad \textit{valid} := \textbf{true};\\
\quad \textbf{for } t \textbf{ from } \textit{start } tn + 1 \textbf{ to } \textit{finish } tn \textbf{ do}\\
\qquad \textbf{if } (\textit{write set of transaction with transaction number } t \textit{ intersects read set})\\
\qquad\quad \textbf{then } \textit{valid} := \textbf{false};\\
\quad \textbf{if } \textit{valid}\\
\qquad \textbf{then } ((\textit{write phase}); tnc := tnc + 1; tn := tnc) \rangle;\\
\quad \textbf{if } \textit{valid}\\
\qquad \textbf{then } (\textit{cleanup})\\
\qquad \textbf{else } (\textit{backup})).
\end{array}
$$

    Things wrapped by a pair of $\langle\rangle$ means a *critical section* that must be protected by a lock
  - Improved serial validation:
    - Move part of validation out of critical section to improve parallelism: for those that committed before I started validation, can validate against them outside of critical section

```
tend := (
    mid tn := tnc;
    valid := true;
    for t from start tn + 1 to mid tn do
    if (write set of transaction with transaction number t intersects read set)
        then valid := false;
⟨finish tn := tnc;
 for t from mid tn + 1 to finish tn do
     if (write set of transaction with transaction number t intersects read set)
       then valid := false;
 if valid
     then ((write phase); tnc := tnc + 1; tn := tnc));⟩
    if valid
        then (cleanup)
        else (backup)).
```

Critical Section

- Parallel validation:
  - Both validation work and commit phase happen outside of critical sections, with the help of maintaining an $active$ set

```
tend = (
⟨finish tn := tnc;
 finish active := (make a copy of active);
 active := active ∪ {id of this transaction});⟩
valid := true;
for t from start tn + 1 to finish tn do
     if (write set of transaction with transaction number t intersects read set)
       then valid := false;
for i ∈ finish active do
     if (write set of transaction Ti intersects read set or write set)
       then valid := false;
if valid
     then (
       (write phase);
       ⟨tnc := tnc + 1;
        tn := tnc;
        active := active—{id of this transaction});⟩
       (cleanup))
     else (
       ⟨active := active—{id of transaction});⟩
       (backup)).
```

Critical Sections

  - Need to validate on both read set and write set for $finish\ active$ because the commit phase is now unprotected from critical section; we need to protect against:
    - interleaved *blind writes* in commit phase
    - data race on same record in commit phase
  - ↑ length of critical section is independent of the number of validating transactions
  - ↓ may lead to unnecessary aborts, due to conflict with an aborted transaction or with someone that's during its commit phase

Silo: Modern OCC

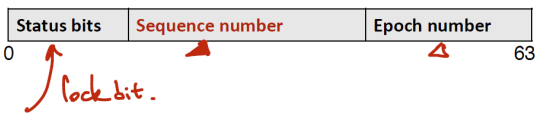
Link: https://dl.acm.org/doi/pdf/10.1145/2517349.2522713

Set-intersection-based validation vs. version-number-based validation:

- In classic OCC, timestamp allocation and critical section locking could become scalability bottleneck on modern multicore machines
- By assigning a *version number* + *lock bit* field to each record, Silo enables fine-grained latching and removes the central timestamp allocation bottleneck

Transaction-ID (TID) word:

- Each record in Silo database contains an extra column of a 64-bit TID word



  - Lock bit acts like a *latch*
  - Sequence number is a version number chosen by the worker thread to be:
    - $>$ TID sequence number of any record seen in the transaction
    - $>$ the worker's most recently chosen TID sequence number
  - Epoch number is for parallel durability logging purposes
- Each write operation during a transaction will lock the record by *atomically* setting the lock bit and updating the sequence number

```
r.lock_bit = 1
r.update_value()
r.update_seq_number()
r.lock_bit = 0
```

- Each read operation could do either thing to ensure reading out consistent bytes atomically:
  - Guard the read with trying to hold the latch
  - Optimistic latching (Silo's approach)

```
do
    tid0 = r.read_TID_word()
    RS[r.key].value = r.value
    tid1 = r.read_TID_word()
while (tid0 != tid1 or tid0.lock_bit == 1)
```

Silo's complete validation & commit protocol:

```
Data: read set R, write set W, node set N,
      global epoch number E
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
compiler-fence();
e ← E;                                  // serialization point
compiler-fence();
// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
          or (record.locked and record ∉ W)
    then abort();
for node, version in N do
    if node.version ≠ version then abort();
commit-tid ← generate-tid(R, W, e);
// Phase 3
for record, new-value in W do
    write(record, new-value, commit-tid);
    unlock(record);
```

- First lock the write set in *sorted* memory address order to prevent deadlocks
- Validation fails if any read record is:
  - modifed since earlier read in execution phase
  - currently locked by another transaction
- It does have to re-read the TID field of every read record, which may be sub-optimal for range queries

Phantom protection mechanisms;

- With 2PL, phantoms can be prevented by *gap locks*, i.e., a lock on a gap between index records
  - Next key lock = index node lock + gap lock before the record
- In Silo, phantoms are checked by validating the versions of accessed index nodes (+ possibly next nodes)

B-link Tree

Link: https://www.csd.uoc.gr/~hy460/pdf/p650-lehman.pdf

B-tree vs. B+-tree vs. B*-tree:

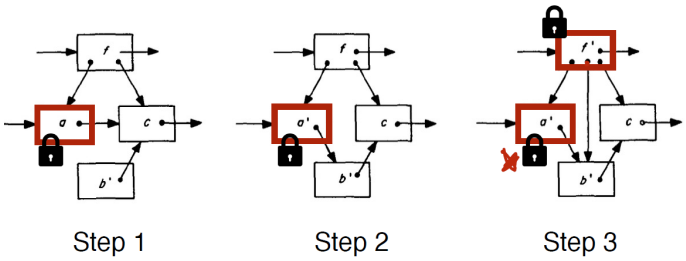
- B-tree:
  - Every node is $2k$ wide and contains k to $2k$ keys (except root), i.e., each node is at least half full
  - All leaf nodes are at the same depth level
  - k is typically large, so fan-out can be large and a lookup traverses a small number of levels
  - Values (or value pointers) stored in all levels
- B+-tree:
  - B-tree, but values (or value pointers) stored only in leaf nodes
  - Leaf nodes are *linked* together to speed up range scans
- B*-tree:
  - B+-tree, and each node is at least $\frac{2}{3}$ full
  - In this paper -- each internal node has a *high key* appended on the right (upper bound on keys)
- Concurrency challenge: these data structures do not natively support concurrent operations
  - Mostly due to the *split* action when a node is full
  - A concurrent insert may split a node and hence mislead a concurrent search to think that a key does not exist, even though it exists but is just in the newly-split *right sibling* node

Traditional lock coupling mechanism:

- A node is *unsafe* if it is full (i.e., contains $2k$ keys)
- *Lock coupling* (aka. *lock crabbing*):
  1. lock parent node
  2. access parent, search for desired child pointer
  3. lock child node
  4. If child node is safe, release parent node
  5. If child node not safe, split it immediately then release parent node
- ↓ root node and higher-level nodes are locked for every operation, becoming a scalability bottleneck
- Lock coupling could also be done optimistically: associate each node's lock bit with a version number; after reading child node version, validate that parent node version has not changed

B-link tree -- add internal node *link* pointers to allow search to find the right node:

- Modifications to B*-tree:
  - Each internal node has a *link pointer* to its right sibling as well
  - Appends a high key to each node, indicating the upper bound of key ranges of the right-most child
- Split procedure:



  - Lock node a to split
  - Create new sibling b, point it to current sibling c
  - Change a's link pointer to b
  - Lock a's parent f, insert new child pointer to b
- Search procedure: if not found in target node but sees that its high key is actually smaller than the search key, it means some split has happened to this node -- follow the link pointer to the right sibling (may repeat)
- In split, how to know who's a's parent?
  - Remember the path walked down the tree during lookup
  - If the parent has been split, will know that by checking high key, so follow its link pointer now to find the real parent node
    - this is the only short period of time when an operation could hold 3 locks simultaneously (a, a's original parent, a's real parent)

Adaptive Radix Tree (ART)

Link: https://db.in.tum.de/~leis/papers/ART.pdf

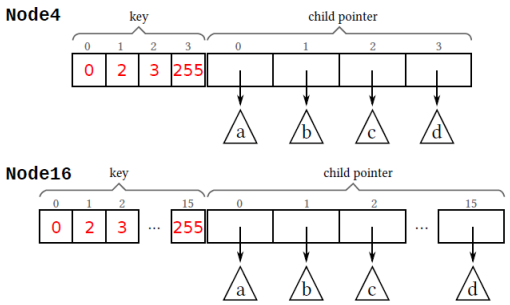
Radix tree (aka. *digital tree* or *prefix tree* or *trie*):

- Path to leaf node implicitly stores the key of length k bits (each edge associated with some bits value)
- *Span*: the number of bits s associated to each node for it to determine the next child
  - Each internal node is then an array of 2^s pointers
  - Extra space required to store internal nodes $\approx \lceil \frac{k}{s} \rceil \times 2^s$
  - Larger span $\rightarrow$ reduced height, but also exponential growth in tree size

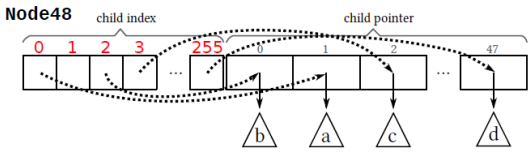
Adaptive radix tree (ART) ideas:

- Using *adaptive* node type according to the number of non-empty slots within it

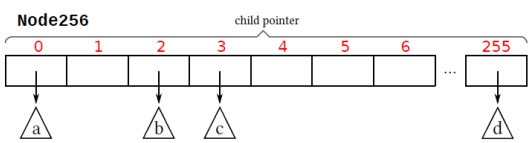
  Examples below assumes span $s = 8$, so each key segment has at most 256 possible values
  - `Node4` & `Node16`



    - key array = array of 4/16 slots, each s bits long storing the key segment if non-empty
    - pointer array = array of 4/16 child pointers corresponding to the key array index
    - Search can be accelerated using SIMD comparison instructions
  - `Node48`



    - key array = 256 slots, storing indices into the child pointer array if that segment value exists
    - pointer array = array of 48 child pointers
    - Why? Because when we have more than 16 keys, a linear search through the key array becomes slower -- better just use the segment value as index (paying slightly higher space usage)
  - `Node256`



    - Simply a flat array of 256 slots, storing child pointer if non-empty
- *Collapsing* internal nodes to reduce redundant information stored
  - *Lazy expansion*: remove internal path to single leaf
    - Expand it on-demand if a conflicting key arrives
    - Requires complete key to be stored somewhere at the leaf
  - *Path compression*: merge one-way node into child node
    - Requires remembering the key segments on the path to merged node in the child node

        - Pessimistic: child node simply allocates an extra buffer for holding the compressed partial key
        - Optimistic: child node only stores the *length* of compressed partial key -- upon access, use that length to consult any leaf node (which stores complete key) in the subtree to get the partial key prefix
        - Hybrid: use constant array to store partial key, switch to optimistic approach if array overflows
    - Lazy expansion can be viewed as a special case of path compression, where the child node is a leaf

ARIES Logging

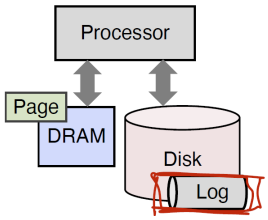
Link: https://cs.stanford.edu/people/chrismre/cs345/rl/aries.pdf

Durability requirements of DBMS:

- The database must recover to a valid state across crashes
- Committed transactions must persist
- Uncommitted transactions must be rolled back

Write-ahead logging (WAL):

- A *log* on persistent storage is the *ground truth* of the database state



    - Any operation that changes state, before reflecting it in-place, must have been durably appended as an entry in the log
    - In some main memory DBMS, the disk contains only the log (+ some *checkpoints* to avoid replaying the entire history after a crash)
- What should be logged for an update depends on the buffer management policy:
    - Stealing --
        - *No steal*: dirty pages stay in DRAM until the transaction commits; those frames holding dirty pages cannot be evicted half way (stolen by other transactions)
        - *Steal*: dirty pages can be flushed to disk before transaction commits
            - ⇒ requires *UNDO logging* before each update
    - Forcing --
        - *Force*: all dirty pages must be flushed at the time of transaction commit
        - *No force*: dirty pages can stay in memory after the transaction commits; allows e.g. batching I/O
            - ⇒ requires *REDO logging* before each update

	Steal	No Steal
Force	**UNDO** only	No REDO nor UNDO
No Force	**REDO** and **UNDO** logging (ARIES)	**REDO** only

Non-volatile memory DB

Disk-based DB **Main memory DB**

    - If both UNDO and REDO logging are required for an update, they may be combined as a single log entry

Deriving *ARIES logging* design:

- Version #1: baseline design
    - Data structures:
        - *Log entry* := `(LSN), txnID, pageID, REDO/UNDO data`; *log sequence number* (LSN) is the offset of log entry in log address space
        - *Data page* := collection of `tuple data`
        - *(Active) transaction table* (TT) := set of `txnID`s
    - Operations:
        - Write -- append REDO/UNDO log entry to log, then update the data page
        - Commit -- append COMMIT entry to log
        - *Recovery* -- after crash, do two passes:
            1. Forward scan entire log:
                - REDO all entries
                - Build the TT -- adding in a `txnID` upon the first time encountering a new transaction's log entry; removing one after seeing a commit
            2. Backward scan entire log: UNDO all entries of uncommitted transactions in TT
    - Limitations:
        - REDO process redoes everything, even those entries that have already been reflected in data pages
        - UNDO process scans entire log again, while many of the early entries do not belong to uncommitted transactions
        - There's no checkpointing mechanism, meaning we need to always start from the beginning of time
- Version #2: add a version number to each data page

  Goal: avoid applying actual REDO if its action has already been reflected on the data page
    - Data structures:
        - *Data page* := `page LSN` + collection of `tuple data`

    - Operations:
        - Write updates page's `pageLSN` to record the latest log entry whose effect has been reflected on the data page
        - REDO phase only actually applies REDO action if entry's `LSN` > page's `pageLSN`
- Version #3: link log entries of the same transaction

  Goal: avoid back-scanning all log entries trying to identifies those belonging to uncommitted transactions

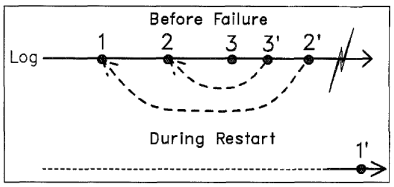
    - Data structures:
        - *Log entry* := `(LSN), txnID, pageID, REDO/UNDO data, prevLSN`
        - *Transaction table* (TT) := set of `txnID, lastLSN`s
    - Operations:
        - Write injects the `prevLSN` of the last log entry belonging to the transaction into every newly-appended log entry
        - REDO phase maintains `lastLSN` for each transaction in TT based on the last update of the transaction
        - UNDO phase then only needs to go through TT, and for each active transaction, start from its `lastLSN` and follow the `prevLSN`s to undo its entries
- Version #4: add checkpointing mechanism

  Goal: avoid starting from the beginning of time for the first scan

    - Data structures:
        - *Dirty page table* (DPT) := set of `pageID, recLSN`s
    - Operations:
        - Add an *Analysis* phase --
            - Builds the TT
            - Also builds the DPT: seeing an update entry to a page puts the page into DPT if not already in DPT, and sets the dirty page's `recLSN`, indicating the first log entry (after the last checkpoint) since when the page becomes "dirty"
        - Periodically *checkpoint* both TT and DPT into log, and clear them after such checkpoint
        - REDO phase starts from the *smallest* `recLSN` in DPT
- Version #5: adopting *compensation log records* (CLRs)

  Goal: in cases of *transaction aborts* or *recursive crashes*, avoid undoing an UNDO recursively

    - Data structures:
        - CLR *Log entry* := `(LSN), txnID, pageID, REDO/UNDO data, prevLSN, undoNxtLSN`
    - Operations:
        - Writing an UNDO entry injects a `undoNxtLsn` pointing to the log entry right before the one that this UNDO is trying to cancel
        - During UNDO phase, do not re-apply UNDO if seeing an CLR, instead just jump to its `undoNxtLSN`



- Overall big picture:
    1. *Analysis phase* -- starting from the last complete checkpoint, scan forward and re-build the TT and the DPT
    2. *REDO phase* -- starting from the smallest `recLSN` in DPT, scan forward and apply REDO entries if the entry's `LSN` > the page's `pageLSN`
    3. *UNDO phase* -- go through the TT, and for each transaction in TT, starting from its `lastLSN` entry, jump backward by following `prevLSN`'s and apply UNDO entries by appending a new CLR entry to log (if seeing CLR during undoing, jump to CLR's `undoNxtLSN`)

Two-Phase Commit (2PC)

Link: https://dl.acm.org/doi/10.1145/7239.7266

Distributed transactions & Atomic commit protocol (ACP):

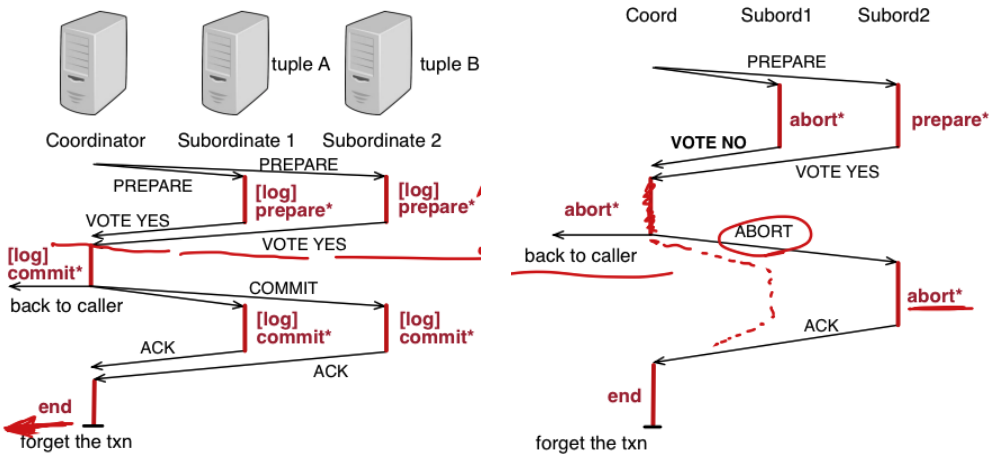
- Data is partitioned; partitions are processed by different servers
- Two traditional architecture: shared-nothing vs. shared-disk
- A *distributed transaction* accesses data across multiple partitions
- An *atomic commit protocol* ensures that all partitions reach the same commit/abort decision of a transaction
    - ⇒ Cannot simply let each node log & commit independently

Two-phase commit (2PC):

- Key idea: assign a *coordinator* node to each transaction, serving as the ground truth of the decision
    - Partition holders that execute the transaction are called *subordinates* or *participants* or *workers*
    - The coordinator does not have to be a standalone node; it could be one of the participant nodes
- Algorithm phases:
    1. Phase 1 -- *Prepare* phase
        1. Coordinator sends `PREPARE` message to participants
        2. Participants execute the transaction, logging any changes made to its partition, plus:
            - If the execution is successful, *logs* `PREPARED`
            - If anything goes unsuccessful and the participant thinks this transaction should abort, *logs* `ABORT`
        3. Participants reply to the coordinator with the result as a *vote*
            - If prepared, reply `VOTE YES`
            - If to abort, reply `VOTE NO`
    2. Phase 2 -- *Commit* phase

1. Coordinator gathers vote replies
    - If all votes are `YES`, the coordinator *logs* the decision of `COMMIT`
    - If any vote is `NO`, the coordinator *logs* the decision of `ABORT`
    - Either way, the coordinator can reply back to the client of this transaction after the logging
2. Coordinator sends final decision (either `COMMIT` or `ABORT`) to participants
    - If the decision is `YES`, send to all participants
    - If the decision is `NO`, only needs to send to participants that voted `YES`
3. Participants send back `ACK` reply upon receiving the final decision
4. Once all `ACK`s are received, the coordinator can now forget about any in-memory state about this transaction

- Vanilla 2PC requires 2 logging delays + 1 network RTT before the transaction's result is sent back to the client caller



Failure situations in 2PC -- use *timeouts* to detect failures:

- Participant timeout waiting for `PREPARE`: self abort
- Coordinator timeout waiting for a `VOTE` reply: decide abort
- Participant timeout waiting for final decision: contact the coordinator or peer participants
    - If the coordinator indeed failed after phase 1 but before sending out any final decisions, 2PC may block indefinitely until the coordinator recovers
    - This is the well-known blocking issue of 2PC; see 3PC for an alternative design that has no blocking, but pays the expensive cost of having one more phase
- Coordinator timeout waiting for an `ACK` reply: contact participants periodically

Optimizations to vanilla 2PC protocol:

- *Presumed Abort* (PA):
    - Always assume that, if no log record is found for a transaction, the decision is abort
        - Therefore, it is safe for participants to send back `ABORT` before logging it, and safe for coordinator to return back to caller before logging the `ABORT` decision
    - *Read-only* participants do not need to log anything and simply just needs to reply `VOTE READ`
        - Completely read-only transactions finishes in just 1 network RTT with no logging at all
- *Presumed Commit* (PC):
    - Force logging a `COLLECTING` on the coordinator before phase 1 starts
        - Then, no need for participants to reply `ACK` at the end of phase 2
    - Not as useful as PA; also, PC & PA are incompatible with each other

Cloud-Native Databases

This section is about recent research progresses in designing distributed database systems that naturally exploit nice features of the cloud. I will only include the abstract of each paper.

Cornus

Link: https://dl.acm.org/doi/10.14778/3565816.3565837

Two-phase commit (2PC) is widely used in distributed databases to ensure atomicity of distributed transactions. Conventional 2PC was originally designed for the shared-nothing architecture and has two limitations: *long latency* due to two eager log writes on the critical path, and *blocking* of progress when a coordinator fails.

Modern cloud-native databases are moving to a storage disaggregation architecture where storage is a shared highly-available service. Our key observation is that disaggregated storage enables protocol innovations that can address both the long-latency and blocking problems. We develop Cornus, an optimized 2PC protocol to achieve this goal. The only extra functionality Cornus requires is an atomic compare-and-swap capability in the storage layer, which many existing storage services already support. We present Cornus in detail and show how it addresses the two limitations. We also deploy it on real storage services including Azure Blob Storage and Redis. Empirical evaluations show that Cornus can achieve up to 1.9X latency reduction over conventional 2PC.

Aria

Link: https://dl.acm.org/doi/10.14778/3407790.3407808

Deterministic databases are able to efficiently run transactions across different replicas without coordination. However, existing state-of-the-art deterministic databases require that transaction read/write sets are known before execution, making such systems impractical in many OLTP applications. In this paper, we present Aria, a new distributed and deterministic OLTP database that does not have this limitation. The key idea behind Aria is that it first executes a batch of transactions against the same database snapshot in an *execution phase*, and then deterministically (without communication between replicas) chooses those that should commit to ensure serializability in a *commit phase.* We also propose a novel deterministic reordering mechanism that allows Aria to order transactions in a way that reduces the number of conflicts. Our experiments on a cluster of eight nodes show that Aria outperforms systems with conventional nondeterministic concurrency control algorithms and the state-of-the-art deterministic databases by up to a factor of two on two popular benchmarks (YCSB and TPC-C).

Amazon Aurora

Link: https://dl.acm.org/doi/10.1145/3035918.3056101

Amazon Aurora is a relational database service for OLTP workloads offered as part of Amazon Web Services (AWS). In this paper, we describe the architecture of Aurora and the design considerations leading to that architecture. We believe the central constraint in high throughput data processing has moved from compute and storage to the network. Aurora brings a novel architecture to the relational database to address this constraint, most notably by pushing redo processing to a multi-tenant scale-out storage service, purpose-built for Aurora. We describe how doing so not only reduces network traffic, but also allows for fast crash recovery, failovers to replicas without loss of data, and fault-tolerant, self-healing storage. We then describe how Aurora achieves consensus on durable state across numerous storage nodes using an efficient asynchronous scheme, avoiding expensive and chatty recovery protocols. Finally, having operated Aurora as a production service for over 18 months, we share the lessons we have learnt from our customers on what modern cloud applications expect from databases.

Snowflake

Link: https://dl.acm.org/doi/10.1145/2882903.2903741

We live in the golden age of distributed computing. Public cloud platforms now offer virtually unlimited compute and storage resources on demand. At the same time, the Software-as-a-Service (SaaS) model brings enterprise-class systems to users who previously could not afford such systems due to their cost and complexity. Alas, traditional data warehousing systems are struggling to fit into this new environment. For one thing, they have been designed for fixed resources and are thus unable to leverage the cloud's elasticity. For another thing, their dependence on complex ETL pipelines and physical tuning is at odds with the flexibility and freshness requirements of the cloud's new types of semi-structured data and rapidly evolving workloads. We decided a fundamental redesign was in order. Our mission was to build an enterprise-ready data warehousing solution for the cloud. The result is the Snowflake Elastic Data Warehouse, or "Snowflake" for short. Snowflake is a multi-tenant, transactional, secure, highly scalable and elastic system with full SQL support and built-in extensions for semi-structured and schema-less data. The system is offered as a pay-as-you-go service in the Amazon cloud. Users upload their data to the cloud and can immediately manage and query it using familiar tools and interfaces. Implementation began in late 2012 and Snowflake has been generally available since June 2015. Today, Snowflake is used in production by a growing number of small and large organizations alike. The system runs several million queries per day over multiple petabytes of data.

In this paper, we describe the design of Snowflake and its novel multi-cluster, shared-data architecture. The paper highlights some of the key features of Snowflake: extreme elasticity and availability, semi-structured and schema-less data, time travel, and end-to-end security. It concludes with lessons learned and an outlook on ongoing work.

FlexPushdownDB

Link: https://dl.acm.org/doi/abs/10.14778/3476249.3476265

Modern cloud databases adopt a *storage-disaggregation* architecture that separates the management of computation and storage. A major bottleneck in such an architecture is the network connecting the computation and storage layers. Two solutions have been explored to mitigate the bottleneck: *caching* and *computation pushdown.* While both techniques can significantly reduce network traffic, existing DBMSs consider them as orthogonal techniques and support only one or the other, leaving potential performance benefits unexploited.

In this paper we present *FlexPushdownDB (FPDB)*, an OLAP cloud DBMS prototype that supports fine-grained hybrid query execution to combine the benefits of caching and computation pushdown in a storage-disaggregation architecture. We build a hybrid query executor based on a new concept called *separable operators* to combine the data from the cache and results from the pushdown processing. We also propose a novel *Weighted-LFU* cache replacement policy that takes into account the cost of pushdown computation. Our experimental evaluation on the Star Schema Benchmark shows that the hybrid execution outperforms both the conventional *caching-only* architecture and *pushdown-only* architecture by 2.2X. In the hybrid architecture, our experiments show that Weighted-LFU can outperform the baseline LFU by 37%.

GPU for DB Analytics

Link: https://dl.acm.org/doi/abs/10.1145/3318464.3380595

There has been significant amount of excitement and recent work on GPU-based database systems. Previous work has claimed that these systems can perform orders of magnitude better than CPU-based database systems on analytical workloads such as those found in decision support and business intelligence applications. A hardware expert would view these claims with suspicion. Given the general notion that database operators are memory-bandwidth bound, one would expect the maximum gain to be roughly equal to the ratio of the memory bandwidth of GPU to that of CPU. In this paper, we adopt a model-based approach to understand when and why the performance gains of running queries on GPUs vs on CPUs vary from the bandwidth ratio (which is roughly 16× on modern hardware). We propose Crystal, a library of parallel routines that can be combined together to run full SQL queries on a GPU with minimal materialization overhead. We implement individual query operators to show that while the speedups for selection, projection, and sorts are near the bandwidth ratio, joins achieve less speedup due to differences in hardware capabilities. Interestingly, we show on a popular analytical workload that full query performance gain from running on GPU exceeds the bandwidth ratio despite individual operators having speedup less than bandwidth ratio, as a result of limitations of vectorizing chained operators on CPUs, resulting in a 25× speedup for GPUs over CPUs on the benchmark.