

Module 1: SQL

Query Syntax:

```
SELECT <columns>
FROM <tbl>
WHERE <predicate>
GROUP BY <columns>
HAVING < predicate>
ORDER BY <columns>
LIMIT <num>
```

Logical Processing Order

- 1. FROM <table>
- 2. [WHERE <predicate>]
- 3. [GROUP BY <column list>]
- 4. [HAVING <predicate>]
- 5. SELECT <column list>
- 6. [DISTINCT]
- 7. [ORDER BY <column list> [DESC/ASC]]
- 8. [LIMIT <amount>]

DBMS may execute a query in an equivalent but different order  
For multi-table queries: perform joins with FROM

**String Comparison:** For LIKE, ‘\_’ means any single character; ‘%’ means zero, one, or more characters

**Aggregation Functions:** SUM, AVG, MAX, MIN, COUNT; can only be used in SELECT, else use subqueries

Subqueries

**Cross-Join:** Cartesian product, combines every row from left table with every row from right table, i.e. ‘FROM a, b’

**Inner Join:** Only outputs shared rows that match on somn column, i.e. ‘FROM a INNER JOIN b ON a.aid = b.bid’

**Left Outer Join:** Outputs shared rows and also left table rows that do not have any matches with the right table; non-matching rows have their columns from the right table filled in with NULL, i.e. ‘FROM a LEFT OUTER JOIN b ON a.aid = b.bid’

**Full Outer Join:** Guarantees that all rows from each table will appear in the output; if a row from either table does not have a match it will still show up in the output and the columns from the other table in the join will be NULL

Module 2: Disk, Buffers, Files

Disk → Page → Record

**Heap File:** No particular ordering of pages or of records; can be implemented with Linked List approach or Page Directory approach;

**Linked List Impl:** Each data page contains records, a free space tracker, and pointers to the next and previous page; one header page acts as start of file and separates the others into full and free pages; when space is needed, empty pages are allocated and appended to free pages portion; when free data pages become full,

they are moved from the free space portion to the front of the full pages portion of the linked list

**Page Dir Impl:** Only uses linked list for header pages; each header page contains a pointer to the next header page and its entries contains both a pointer to a data page and teh amount of free space left in the data page; since header pages’ entries store pointers to data; the data itself no longer needs to store pointers to neighboring pages; ihserting is much faster

**Sorted File:** pages are ordered and records within each page are sorted by key(s); implemented using page directories and enforce an ordering upon data pages; searching takes LogN IOs and insertion on average takes logN + N (N/2 to read and N/2 to write or shift back)

**FLR vs VLR:** FLR only contains fixed length fields (int, bool, date) and FLRs w/ same schema have same num of bytes; VLRs contain both fixed and variable length fields (e.g. varchar), resulting in each VLR having potentially diff num of bytes; VLRs store all fixed length fields before variable length fields and use a record header that contains pointers to the end of the variable length fields; every record is uniquely identified by its record id, (page num, record num on page)

**Pages w/ Fixed Length Records:** Pages with FLRs always use page headers to store the num of records currently on the page; if page is packed, there is no gaps btwn records; insertion is easy as we can calculate next avail position using num of existing reocrds and length of each record; if page is unpacked, page header stores additional biptmap that breaks page into slots and tracks which slots are open/taken

**Pages w/ Variable Length Records:** Each page has a page footer that maintains a slot directory tracking slot count, a free space pointer, and entries; slot count tracks num of filled/empty slots; free space pointer points to next free position within page; each entry in slot directory consists of a (record ptr, record length) pair; if page is unpacked, deletion involves finding record’s entry within slot directory and settting both record ptr and record length to null; For future insertions, the record is inserted into the page at the free space pointer and a new [pointer, length] pair is set in any available null entry. In the case where there are no null entries, a new entry is added to the slot directory for that record. The slot count is used to determine which offset the new slot entry should be added at, and then the slot count is incremented. Periodically, records will be reorganized into a packed state where deleted records are removed to make space for future insertions.; For future insertions, the record is inserted into the page at the free space pointer and a new [pointer, length] pair is set in any available null entry. In the case where there are no null entries, a new entry is added to the slot directory for that record. The slot count is used to determine which offset the new slot entry should be added at, and then the slot count is incremented. Periodically, records will be reorganized into a packed state where deleted records are removed to make space for future insertions.

Module 3: Indices and B+ Trees

**Property 1:** The number d is the order of a B+ tree. Each node (with the exception of the root node) must have  $d \leq x \leq 2d$  entries assuming no deletes happen (it’s possible for leaf nodes to end up with j d entries if you delete data). The entries within each node must be sorted.

**Property 2:** In between each entry of an inner node, there is a pointer to a child node. Since there are at most 2d entries in a node, inner nodes may have at most 2d + 1 child pointers. This is also called the tree’s fanout

**Property 3:** The keys in the children to the left of an entry must be less than the entry while the keys in the children to the right must be greater than or equal to the entry.

**Property 4:**All leaves are at the same depth and have between d and 2d entries (i.e., at least half full)

**Search:** Execute binary serach on each page for the node split, then follow pointer to next node

**Insertion:** 1. Find leaf node L in which value goes into. Traverse down the tree and add the key and record to the leaf node in order. 2. If L overflows (has more than 2d entries), (2a) split L into  $L_1$  and  $L_2$  and keep d entries in  $L_1$  and d+1 entries in  $L_2$ . (2b) if L was leaf, copy  $L_2$ ’s 1st entry to parent node; if L was not leaf, move  $L_2$ ’s 1st entry into the parent; (2c) adjust pointers; 3. If parent overflows, then recurse 2. on parent (will increase tree height)

**Counting IOs:** 1. Read appropriate root-to-leaf path 2. Read appropriate data page(s); if multiple, allocate a read IO for each page; account for clustering for Alt. 2 or 3; 3. Write data page(s), if modifying it; 4. update index page(s) (leaf or inner node).

**Counting Example:** For a alt-2 unclustered B+ tree, we want to delete the only 11-year-old from the database. 1 IO for each of teh 2 relevant index pages plus 1 IO to read the dat apge where teh record is (once in memory, we can delete record from page) plus 1 IO to write the modified data page back to disk plus 1 IO to write the modified leaf page to disk.

**Alt versions:** 1. leaf pages are disk pages, contain records; 2. leaf pages hold pointesr to corresp. records; 3. leaf pages hold lists of ptrs to records

**Clustered:** Data pages are sorted on same index on which the B+ tree is built; cheaper IO cost comes from caching as two records with close keys likely will be in teh same page, so the second one can be read from the cached page; just need one page to get all records that have a similar key; 1 I/O per page of records rather than 1 I/O per record

**Bulk Loading:** 1. Sort data on key the index wil be built on; 2. Fill leaf pages until some fill factor  $f$ ; 3. Add ptr from parent to leaf page. if the parent overflows, follow process similar to insertion; split parent into two nodes and keep  $d$  entries in  $L_1$  and since a parent node overflowed, move  $L_2$ ’s first entry into the parent; 4. adjust pointers;

Module 6: Iterators and Joins

**Iterators:**  
 setup(), init(), next(), close()  
 useful because it streams data and therefore does not have to use  
 a lot of memory or space to access large amounts of data that  
 might not fit into memory

**Cost Notation for Join Analysis:**  
 [R]: the number of pages to store R  
 p<sub>R</sub>: the number of records per page of R  
 |R|: the cardinality (number of records) of R (= [R] · p<sub>R</sub>)

**Variants of Nested Loop Join:**  
*Simple:* [R] + |R|[S]  
 We can change join order to lower cost.  
 for each record *r* in R do  
   for each record *s* in S do  
     if  $\theta(r, s)$ , then add  $\langle r, s \rangle$  to result buffer  
*Page:* [R] + [R][S]  
 We can operate at granularity of pages rather than tuples  
 for each rpage in R:  
   for each spage in S:  
     for each rtuple in rpage:  
       for each stuple in spage:  
         if  $\theta(rtuple, stuple)$ :  
           add  $\langle rtuple, stuple \rangle$  to result buffer  
*Block:*  $[R] + \lceil \frac{[R]}{B-2} \rceil [S]$   
 We can extend granularity to blocks of pages.  
 for each rchunk of B-2 pages of R  
   for each spage of S:  
     for all matching tuples in spage and rchunk:  
       add  $\langle rtuple, stuple \rangle$  to result buffer

**Index Nested Join Loop:**  
 We join one data table with another data table that is indexed on  
 a B+ tree.  
 for each rtuple in R:  
   for each stuple in S where rtuple == stuple:  
     add  $\langle rtuple, stuple \rangle$  to result  
 If index uses Alt. 1, the cost is just that to traverse tree from root  
 to leaf. For Alt. 2 and 3, the total cost is the sum between the  
 cost to look up the record id (tree height) plus the cost to retrieve  
 the record having found the record id. The retrieval cost is 1 I/O  
 per page of matching S tuples if the index is clustered; if index is  
 unclustered, 1 I/O per matching S tuple.  
*Unclustered Cost*(*R*, *S*) = [R] + |R| ·  
 (num matching stuples for each rtuple) · (access cost per stuple).  
*Clustered Cost*(*R*, *S*) = [R] + |R| ·  
 (num matching spages for each rtuple) · (access cost per spage).

**Sort Merge Join:**  
 1) Sort tuples in R and S by join key  
 2) Merge-scan the sorted partitions and output matches  
 if (!mark)  
   while (r < s) advance r  
   while (r > s) advance s  
   mark = s // mark start of "block" of S  
 if (r == s)  
   result =  $\langle r, s \rangle$

advance s  
 yield result  
 else  
   reset s to mark  
   advance r  
   mark = NULL  
*average cost* = sort R + sort S + ([R] + [S])

**Grace Hash Join**  
 1) Partition tuples from R and S by join key and store on disk.  
 2) Build + Probe a separate hash table for each partition.  
 for relation Cur in R, S // stage 1  
   for page in Cur  
     read page into input buffer  
     for tuple on page  
       place tuple in output buf hash<sub>p</sub>(tuple.joinkey)  
       if output buf full, flush to disk partition  
   flush remaining output bufs to disc partitions  
 for *i* in [0...(B – 1)]  
   for page in R<sub>*i*</sub>  
     for tuple on page  
       build tuple into memory hash<sub>r</sub> (tuple.joinkey)  
   for page in S<sub>*i*</sub>  
     read page into input buffer  
     for tuple on page  
       prob memory hash<sub>r</sub> for matches  
       send all matches to output buffer  
       flush output buffer if full

Module 7: Query Optimization

**Common Heuristics**  
 Apply *selections* as long as you have the relative columns.  
*Project* unnecessary columns away as soon as we can, so extra  
 data does not get copied or moved around unnecessarily.  
 We want to minimize *cartesian products* since they are very costly.  
 Selection can be cascaded and is commutative.

**Physical Equivalences**  
*Base Table Access:* heap scan and index scan.  
*Equijoin:* Block Nested Loop for simple operations that exploit  
 extra memory, Index Nested Loop is effective if 1 table is  
 relatively small and the other is indexed properly, Sort-Merge  
 Join is good with small memory, equal-size tables, Grace Hash  
 Join is even better than sort if the tables are differently sized.  
*Non-Equijoins:* Block Nested Loop is the only option, as the  
 others are designed for finding matches.

Module 8: Transactions and Concurrency

**Definition and Advantages**  
 Multiple transactions are allowed to run concurrently in the  
 system. Increased processor/disk utilization means greater  
*throughput* (transactions per second), and one transaction's  
*latency* (response time per transaction) is not dependent on  
 another unrelated transaction.

**Lost Update:** Two users update the record at the same time.  
**Inconsistent Read:** User reads in the middle of another user's  
 transaction which is not a state intended by either user.

**Dirty Read:** A user updates the record while another user reads  
 the record.

**Atomicity:** All actions in the Xact happen, or none happen.  
 DBMS should ensure that updates of a partially executed  
 transaction are not reflected.

**Consistency:** If the DB starts out consistent, it ends up  
 consistent at the end of Xact (primary key constraints, type  
 constraints, or functional dependencies like foreign keys).

**Isolation:** Execution of each Xact is isolated from that of others.  
 Each Xact executes as if it ran by itself. The net effect should be  
 identical to executing all transactions in some serial order.

**Durability:** If a Xact commits, effects persist. Any commits not  
 yet sent to disk must survive any failures or system crashes.

DBMS typically ensures *atomicity* and *durability* by *loggingg* all  
 actions: it undoes the actions of failed/aborted transactions and  
 redoes actions of committed transactions not yet propagated to  
 disk when the system crashes.

**Transaction Schedule:** A sequence of actions on data from one  
 or more transactions. Actions include Begin, Read, Write,  
 Commit, and Abort. By convention, we only include committed  
 transactions and omit Begin and Commit.

**Serial Equivalence / Serializability:** 2 schedules are  
 equivalent if they involve the same transactions, each individual  
 transaction's actions are ordered the same, and both schedules  
 leave the DB in the same final state. We want to find an optimal  
 schedule that is equivalent to a *serial* schedule where each  
 transaction runs from start to finish without any intervening  
 actions from other transactions.

Two operations **conflict** when they 1) are by different  
 transactions, 2) are on the same resource, and 3) at least one of  
 them is a write.

Two schedules are **conflict equivalent** iff they involve the same  
 actions of the same transactions and every pair of conflicting  
 actions is ordered the same way.

Schedule S is **conflict serializable** if S is conflict equivalent to  
 some serial schedule. This implies S is also serializable. However,  
 note that some serializable schedules are not cconflict serializable.  
 So when we test if a schedule is conflict serializable, we may  
 produce false negatives for whether it is also serializable. This is  
 the cost of a conservative, yet highly more efficient, test. We can  
 transform S into a a serial schedule by swapping consecutive  
 non-conflicting operations of different transactions.

**Conflict Dependency Graph:** There is one node per Xact.  
 There exists an edge from *T<sub>i</sub>* to *T<sub>j</sub>* if an operation *O<sub>i</sub>* conflicts  
 with an operation *O<sub>j</sub>* and *O<sub>i</sub>* appears earlier in the schedule than  
*O<sub>j</sub>*. The schedule is conflict serializable iff its dependency graph is  
 acyclic.

**View Serializability:** An alternative notion of serializability

that offers fewer false negatives. Schedules S1 and S2 are view equivalent if:

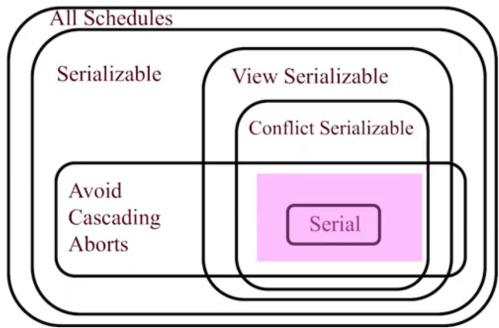
- 1) *Same initial reads*: If  $T_i$  reads initial value of A in S1, then  $T_i$  also reads initial value of A in S2.
- 2) *Same dependent reads*: If  $T_i$  reads value of A written by  $T_j$  in S1, then  $T_i$  also reads value of A written by  $T_j$  in S2.
- 3) *Same winning writes*: If  $T_i$  writes final value of A in S1, then  $T_i$  also writes final value of A in S2.

**Two Phase Locking (2PL)**: Most common scheme for enforcing conflict serializability; Considered "pessimistic" - will set locks on resources for fear of conflict; this pessimism induces some cost; An Xact must obtain a S (shared) lock before reading and an X (exclusive) lock before writing.

**2PL acquires and releases**. A transaction cannot acquire any new locks after it begins releasing locks (this point is the "lock point"). Lock points guarantee conflict serializability because at these points, transactions have everything they need locked and any other conflicting transactions either started their release phase before this point or are blocked waiting for this transaction.

**Cascading Aborts**: The abort/rollback of one transaction requires abort/rollback of another transaction. Example:  $R_1(A), W_1(A), R_2(A), W_2(A), Abort_1$ .

**Strict 2PL**: Same as 2PL, but only when a transaction completes (either commits all writes or aborts and undoes all writes) does it release all its locks (thereby avoiding cascading aborts)



**Lock Management**: When lock request arrives, it either a) does not conflict with locks held by other transactions and is put in the granted set or b) does conflict and is put into the wait queue (FIFO).

**Deadlock**: A cycle of Xacts waiting for locks to be released for each other.

We can **avoid deadlocks** with two approaches: Given  $T_i$  wants a lock that  $T_j$  holds, 1) *Wait-Die*: if  $T_i$  has higher priority,  $T_i$  waits for  $T_j$ ; else  $T_i$  aborts; 2) *Wound-Wait*: 1) if  $T_i$  has higher priority,  $T_j$  aborts; else  $T_i$  waits

Deadlock avoidance succeeds b/c the same priority scheme applies on all lock conflicts, DBMSs can safely roll back transactions, and higher-priority transactions eventually get access.

**Deadlock Detection and Waits-For Graph**: An arrow from a  $T_i$  node to a  $T_j$  node signifies  $T_i$  waits for  $T_j$ , and a cycle in the graph means a deadlock exists. A DBMS will periodically extract a waits-for graph, find cycles, and abort one of the deadlocked transaction on the cycle.

**Additional Lock Modes**: **IS**: Intent to get S lock(s) at finer granularity, **IX**: Intent to get X lock(s) at finer granularity, **SIX**: S and IX at the same time

**Multigranularity Locking Protocol**: To get S or IS lock on a node, must hold IS or IX on parent node; to get X or IX or SIX on a node, must hold IX or SIX on parent node; must release locks in bottom-up order

**Compatibility Between Locks**:

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

### Module 9: Recovery

**Sources of Aborted Xacts** 1) User explicitly issues rollback; 2) Deadlock; 3) System crash; 4) Consistency, i.e. integrity constraint violation

**Sources of DBMS Crashes** 1) Operator Error; 2) Configuration Error; 3) Soft/Hardware failure

**Force / No Force**: Ensures durability; When a Xact has finished, *Force* pushes all modified data pages to disk before the Xact commits. Could be unnecessary because since other Xacts likely also are modifying the in-memory page, we write to disk repeatedly; *No-Force* writes to disk only when the page needs to be evicted from the buffer pool; complicates durability; if DB crashes after transaction commits, dirty pages may not be written to disk and are lost from memory but will be resolved by redoing operations during recovery.

**Steal / No Steal**: Ensures atomicity; *No-Steal* does not allow pages to be evicted from memory (written to disk and saved) until the Xact commits; *Steal* allows modified pages to be written to disk before a Xact finishes; the DB can *steal* pages before a Xact has even finished; complicates atomicity the DB is left in an intermediate state) but will be resolved by undoing bad operations during recovery.

**ARIES - Analysis**: Rebuild what the Xact Table and the Dirty Page Table looked like at the time of the crash. Scan all log records from the start.

1. On any record that is not an END record: add the Xact to the Xact Table (if necessary). Set the lastLSN of the Xact to the LSN

- of the record you are on.
2. If the record is a COMMIT or an ABORT record, change the status of the Xact in the Xact table accordingly.
3. If the record is an UPDATE record, if the page is not in the DPT add the page to the DPT and set recLSN equal to the LSN.
4. If the record is an END record, remove the Xact from the Xact Table.

**Checkpointing**: Writes the contents of the Xact Table and the DPT to the log in order to speed up the analysis phase. Rather than starting at the beginning of the log when we analyze, we can start at the last checkpoint. Writes two records to the log, a *< BEGINCHECKPOINT >* record and an *< ENDCHECKPOINT >* record that says when we finished writing the tables to the log.

**ARIES - Redo**: Ensures durability. Starts at the smallest recLSN in the DPT b/c that is the first op that may not have made it to disk. Will redo all UPDATE and CLR ops unless one of the following conditions is met:

1. The page is not in the DPT. If the page is not in the DPT, it implies that all changes have already been flushed to disk.
2.  $recLSN \geq LSN$ . The first update that dirtied the page occurred after this operation. Implies that the operation we are currently at has already made it to disk, other wise it would be the recLSN.
3.  $pageLSN(disk) \geq LSN$ . If the most recent update to the page that made it to disk occurred after the current operation, then we know the current operation must have made it to disk.

**ARIES - Undo**: Ensures atomicity; starts at the end of the log; undoes every UPDATE for each Xact that was active (running/aborting) at the time of the crash so we do not leave the state in an intermediate state. Will not undo UPDATE if it already has been undone and has a CLR record already in the log for that UPDATE. For every UPDATE undone, we write a corresponding CLR record to the log. It will have an additional field, undoNextLSN, that stores the LSN of the next operation to be undone for that Xact (coming from the prevLSN of the operation that is being undone). Once all undoing operations for a Xact are completed, write the END record for that Xact to the log.

### Module 11: Parallel Query Processing

### Module 12: Distributed Transactions