DB FINAL

IO: an I/O is any time we either write a page from memory to disk or read a page from disk into memory.

SQL

Relational Terminology

- **Database**: Set of named Relations
- **Relation (Table)**: ***Schema***: description ("metadata"); ***Instance***: set of data satisfying the schema
- **Attribute**: Column, Field; **Tuple**: Record, row

SQL Language

DDL: Data Definition Language: Define & modify schema
DML: D Manipulation L: Queries can be written intuitively
Primary Keys (DDL): Primary Key column(s) provides a unique "lookup key" for the relation, cannot have any duplicate values, can be made up of >=1 column
Aggregates: Produces 1 row of output
GROUP BY: can put grouping columns in SELECT list
HAVING: applied after grouping and aggregation. HAVING can only be used in aggregate queries.
HAVING 是对组进行判断 - WHERE 是对行进行判断
Join: inner/natural join 只保留相等项(交集), left outer join保留相等项和左余集, full outer join保留所有(行中可有NULL).
Combining Predicates:

- **Boolean logic** in WHERE (i.e., AND, OR)
- **Set operations**: Intersect 交, Union 并, Except 差 全部默认不带 ALL 操作，ALL会保留重复值

NULL: NULL op NULL is NULL; WHERE NULL: do not send to output; Boolean Logic: N and T is N, N and F is F, N or T is T, N or F is N. Aggregates ignore NULL inputs.

Disk, Buffers and Files

Disks

Hierarchy: Disk→SSD→RAM→L1/2Cache→Register
Lower I/O cost: reduce seek time and rotational delays. -> Arrange file pages sequentially by 'next' on disk
'Next' block: sequential blocks on same track, followed by; blocks on same cylinder, f; blocks on adjacent cylinder.
Disk Space Management: Lowest layer of DBMS. Provides "next" locality and abstracts FS/device details.

DB File Structure

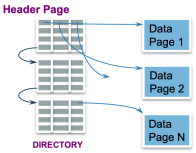
DB FILE: A collection of pages, each containing a collection of records.

- **Unordered Heap Files**: Records placed arbitrarily across pages
- **Clustered Heap Files**: records and pages are grouped
- **Sorted Files**: Pages and records are in sorted order
- **Index Files** 检索的文件系统: (B+ Trees, Linear Hashing) Contain records or point to records in other files (指针)

Record id: a pointer encoding pair of (**pageID**, **location** on page). Insert/delete/modify records.

Page Header

Header: may contain: Number of records; Free space; Maybe a next/last pointer; Bitmaps, Slot Table.
Options for Page Layouts depend on: Record length (fixed or variable); Page packing (packed or unpacked).
Page Directory: 存在RAM里，多个标签页，每个目录单独指向data page. **I/O cost** to insert a record into the directory heap file in the best case: 1 (read header page) + 1 (read data page) + 1 (write updated data page) + 1 (write updated header page) = 4 I/Os.



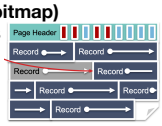
Page Layout

Fixed Length Records: Packed

- Easy to add: just append
- Delete: Packed implies re-arrange! Record Id pointers need to be updated -> expensive
- Record id = (pageId, record number in page)

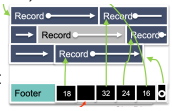
Fixed Length Records: Unpacked (bitmap)

- Bitmap denotes "slots" with records
- Record id: record number in page
- Insert: find first empty slot
- Delete: Clear bit
- Record id: (Page 2, Record 4)



Variable Length Records: Slotted Page

- Delete: Set slot directory pointer to null, doesn't affect other pointers
- Insert:
  - Place record in free page space
  - Create pointer/length pair in next open slot in slot directory
  - Update the free space pointer
- Growing:
  - Tracking number of slots in slot directory
  - Extend slot directory: Slots grow from end of page inward; Records grow from beginning of page inward.
  - And update count

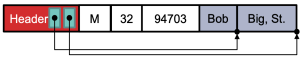


Record Layout

Record Formats: Fixed Length
Finding i'th field: done via arithmetic (fast). Not compact.

Record Formats: Variable Length

- Idea: move variable length fields to end -> fast access
- Introduce a record header
- Advantages: Direct access & no "escaping"; Handle null fields easily; useful for fixed length records too!



File Organizations

Heap: Insert appends to end of file; **Sorted**: File compacted after deletion, sorted according to search key.
Cost of Operations: B: #data blocks; R: #records per block; D: avg time to read / write disk block

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	($\log_2$B)*D
Range Search	B*D	(($\log_2$B)+pages)* D
Insert	2*D	(($\log_2$B)+B)*D
Delete	(0.5*B+1)*D	(($\log_2$B)+B)*D

- Can we do better? -> Indexes!

Index and B+ Trees

Index: a data structure that enables fast lookup and modification of data entries by search key.

- Lookup: may support many different operations
  - Equality, 1-d range, 2-d region, ...
- Search Key: any subset of columns in the relation
  - Do not need to be unique, e.g. (first-name) or (first-name, last-name)
- Data Entries: items stored in the index
- Modification: want to support fast insert and delete
- Types of indexes exist: B+-Tree, Hash, R-Tree, GiST, ...

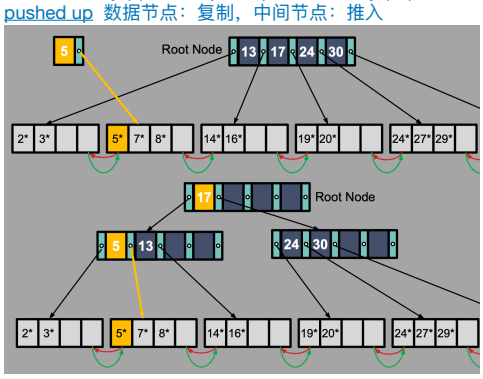
B+ Tree: Dynamic Tree Index, Always Balanced, Support efficient insertion & deletion, Grows at root not leaves!
Key Invariant: Keys in the children to the left of an entry must be less than the entry while the keys in the children to the right must be greater than or equal to the entry.
Occupancy Invariant: Each interior node is at least partially full. The root is special.

- d <= #entries <= 2d, d: order of the tree
- max fan-out = 2d+1
- eg: how big is a height 3 B+ tree? (d=2)
  - fanout = 2d+1 = 5
  - 5^3x4 = 500 records

Disk Layout: All in a single file, Data Pages first.
Pointers: Data pages at bottom need not be stored in logical order -> next and prev pointers
Search: starting at root, complexity $O(\log_F(\#\text{Pages}))$.
Insert:

- Find the correct leaf
- If there is room in the leaf
  - Just add the entry
  - Sort the leaf page by key
- If there is not enough room: Split leaf
  - Redistribute entries evenly
  - Fix next/prev pointers
  - Copy up from leaf the middle key
  - No room in parent? Recursively split index nodes
    - Redistribute the rightmost d keys
    - Push up middle key
- Check invariants
  - Key invariant
  - Occupancy Invariant

The **leaf** entry (5) was copied up, **index** entry (17) was pushed up 数据节点：复制，中间节点：推入



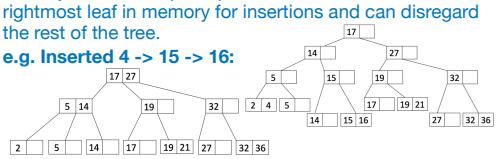
B+ Tree Insert: Algorithm Sketch

- Find the correct leaf L.
- Put data entry onto L.
  - If L has enough space, done!
  - Else, must split L (into L and a new node L2)
    - Redistribute entries evenly, copy up middle key
    - Insert index entry pointing to L2 into parent of L.
  - Step 2 can happen recursively
    - To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits)

Tree growth: gets wider or one level taller at top.
Splits "grow" tree; root split increases height.
Counting IOs: general procedure

1. Read the appropriate root-to-leaf path.
2. Read the appropriate data page(s). If we need to read multiple pages, we will allot a read IO for each page.
3. Write data page, if you want to modify it. Again, if we want to do a write that spans multiple data pages, we will need to allot a write IO for each page.
4. Update index page(s).

Total capacity of height h, order d: $(2d)(2d+1)^h$ entries.
Alternative 1: the leaf pages are the data pages. Rather than containing pointers to records, the leaf pages contain the records themselves. **Alternative 2**: the leaf pages hold pointers to the corresponding records. **Alternative 3**: the leaf pages hold lists of pointers to the corresponding records. This is more compact than Alternative 2 when there are multiple records with the same leaf node entry.
Bulk-loading: to reduce the number of IOs required to construct a B+ tree initially. When bulk loading B+ trees, we only need to keep the path from the root to the rightmost leaf in memory for insertions and can disregard the rest of the tree.
e.g. Inserted 4 -> 15 -> 16:



Buffer Manager

Two questions: Handling dirty pages; Page Replacement.
Page pin count: 该 page 正在被调用几次 The pin count is decremented by whoever requested the page to signify that they are done with it.
When a Page is Requested ... (Steps)

- If (requested page is not in pool):
  - Choose an un-pinned (pin_count = 0) frame for replacement.
  - If frame dirty, write current page to disk, mark "clean"
  - Read requested page into frame
- Pin the page and return its address

LRU

- Least Recently Used (LRU)
  - Pinned Frame: not available to replace
  - Track time each frame last unpinned (end of use)
  - Replace the frame which was least recently used
- Very common policy: intuitive and simple
  - Good for repeated accesses to popular pages (temporal locality)
  - Can be costly. Need to "find min" on the last used attribute (priority heap data structure), O(n)

Approximate LRU: CLOCK policy

- If current frame has pin-count > 0: Skip
- If current frame not pinned, Ref bit set:
  - Clear ref bit
  - Skip
- If current frame not pinned, Ref bit unset:
  - Replace
  - Set pinned
  - Set ref bit
  - Advance clock

解除使用的同时设置参考位为 1
Might perform poorly when repeated scans of big files.
The clock algorithm is actually a FIFO algorithm with a second chance policy.
Reset the reference bit to 1 if we get a hit on a page where its reference bit is now 0 since it has recently used.

MRU

便于大量顺序访问: In limit: ~(B-1)/N hit rate
Improvement for sequential scan: prefetch

Relational Algebra

$$\pi_{S.name}(\sigma_{R.bid=100 \wedge S.rating>5}(R \bowtie_{R.sid=S.sid} S))$$

Unary Operators: on single relation

- Projection (π): 保留列 SELECT (会重复)
- Selection (σ): 保留行 WHERE
- Renaming(ρ): 改名 AS

Binary Operators: on pairs of relations

- Union (∪): 没有重复项 结合的两个表需列相同(schema同)
- Set-difference (—): EXCEPT
- Cross-product (×): 两个表无须兼容

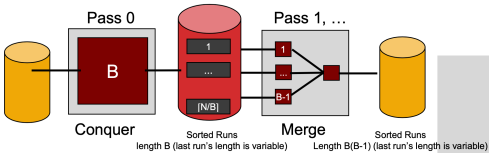
Compound Operators: 混合运算符

- Intersection(∩): 保留列 SELECT
  - S1 ∩ S2 = S1 – (S1 – S2)
- Joins(⋈_θ, ⋈): 叉乘的高效实现

Extended Relational Algebra: Group By / Aggregation Operator (γ): HAVING

Sorting and Hashing

Sorted run: A sorted run is any sequence of pages that is sorted.



General External Merge Sort

- More than 3 buffer pages
- To sort a file with N pages using B buffer pages:
  - Pass 0: use B buffer pages. Produce |N/B| sorted runs of B pages each.
  - Pass 1, 2, ..., etc.: merge B-1 runs at a time.
- **Cost**: overall sort take $1 + \lceil log_{B-1}\lceil N/B \rceil \rceil$ passes, and thus $2N * (1 + \lceil log_{B-1}\lceil N/B \rceil \rceil)$ I/Os.
- Each pass takes 2N **IOs** where N is the total number of data pages because each page gets read and written in a pass.
- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each
    - last run is only 3 pages
  - Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each
    - last run is only 8 pages
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages
- **Memory Requirement**: How big of a table can we sort in 2 passes: B(B-1). In p passes: B(B-1)^(p-1).

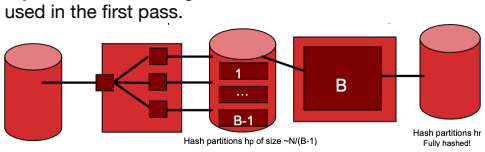
Hashing

Hashing in DB: In a database, grouping like values together is called hashing.

Divide and Conquer: The "divide" phase will be partitioning passes, and the "conquer" phase will be actually constructing the hash tables.

Partition: The most important property of each partition: if a certain value appears in that partition, all occurrences of that value in our data appear in that partition.

Fitting in memory: the partition must be B pages big or less. For the partitions that are too big, we simply **repartition** them using a different hash function than we used in the first pass.



Cost: the only reliable way to count the number of I/Os is to go through each pass and see exactly what will be read and what will be written. Let m be the total number of partitioning passes required, let r_i be the number of pages you need to read in for partitioning pass i, let w_i be the number of pages you need to write out of partitioning pass i, and let X be the total number of pages we have after partitioning that we need to build our hash tables out of. Here is a formula for the number of I/Os:

$$(\sum_{i=1}^{m} r_i + w_i) + 2X$$

The final 2X part, says that in order to build our hash tables, we need to read and write every page that we have after the partitioning passes.

How big of a table can we sort in 2 passes: B(B-1).

Iterations and Joins

R: the number of pages to store R

P_R: number of records per page of R

|R|: the cardinality (number of records) of R

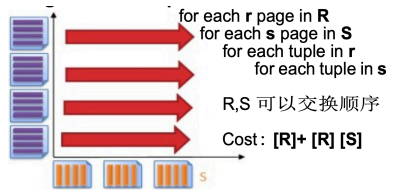
Simple Nested Loops Join

```
for each record r_i in R:
    for each record s_j in S:
        if θ(r_i,s_j):
            write <r_i, s_j> to disk
```

Take each record in R and read in every single page in S searching for a match.

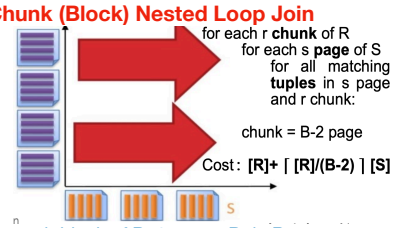
The I/O cost of this would then be [R]+|R|[S] or swap RS.

Page Nested Loop Join



For a page of R, take all the records and match them against each record in S, and do this for every page of R. We have B buffer pages, but our algorithm only uses 3 – one for R, one for S, and one for the output buffer.

Chunk (Block) Nested Loop Join




```
for each block of B–2 pages Br in R:
  for each page ps in S:
    for each record ri in Br:
      for each record sj in ps:
        if θ(ri,sj):
          yield <ri, sj>
```

Now, we're taking advantage of our B buffer pages to reduce the number of times we have to read in S.

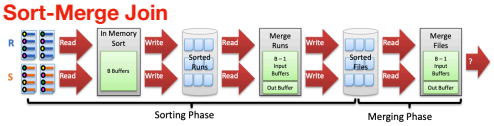
Index Nested Loops Join

Cost = [R] + |R| * cost to find matching S tuples

```
for each record ri in R:
  for each record sj in S where θ(ri,sj)==true:
    yield <ri, sj>
```

If we have an index on S that is on the appropriate field (i.e. the field we are joining on), it can be very fast to look up matches of ri in S. The cost to look up matching records in S will differ based on the type of index.

Sort-Merge Join



仅适用 Equi/ Natural Joins

Average **I/O cost**: cost to sort R + cost to sort S + ([R]+[S]) (though it is important to note that this is not the worst case!->last term should be |R|*[S]).

$B > \sqrt{\max([R], [S])}$ ⇒ sort R & S in 2 passes

改进：combines last pass of merge-sort with join pass

An important refinement: You can combine the last sorting phase with the merging phase, provided you have enough room in memory to allocate a page for each run of [R] and for each run of [S]. The final merge pass is where you allocate a page for each run of R and each run of S. This process save 2∗([R] + [S]) I/Os.

Naive Hash Join

仅适用 Equi/ Natural Joins

Even if we don't have an index, we can construct a hash table that is B-2 pages big (We need one page for the current page in S and one page to store output records) on the records of R, fit it into memory, and then read in each record of S and look it up in R's hash table to see if we can find any matches on it. **Cost**: [R] + [S] I/Os.

Problem: this relies on R being able to fit entirely into memory (specifically, having R being ≤ B – 2 pages big).

Grace Hash Join

仅适用 Equi/ Natural Joins

To fix the problem of NHJ, we repeatedly hash R and S into B-1 buffers so that we can get partitions that are ≤ B – 2 pages big, enabling us to fit them into memory and perform a Naive Hash Join. More specifically, consider each pair of corresponding partitions Ri and Si (i.e. partition i of R and partition i of S). If Ri and Si are both > B-2 pages big, hash both partitions into smaller ones. Else, if either Ri or Si ≤ B-2 pages, stop partitioning and load the smaller partition into memory to build an in-memory hash table and perform a Naive Hash Join with the larger partition in the pair.

I/O cost: the cost of hashing + the cost of Naive Hash Join on the subsections.

The cost of hashing can change based on how many times we need to repeatedly hash on how many partitions. The cost of hashing a partition P includes the I/O's we need to read all the pages in P and the I/O's we need to write all the resulting partitions after hashing partition P.

2 Pass: (Appendix. 3)

- Partitioning phase: R+W both relations: 2([R]+[S]) I/Os
- Matching phase: R both relations, forward output: [R]+[S]
- Total cost of 2-pass hash join = 3([R]+[S])

Query Optimization

Query Optimization: System R Optimizer

- **Plan Space**: Too large, must be pruned.
  - Algorithmic insight:
    - Many plans have the same "overpriced" subtree
    - Ignore all those plans
  - Common heuristic: consider only left-deep plans
  - Common heuristic: avoid Cartesian products
- **Cost estimation**
  - Very inexact, but works ok in practice.
  - Stats in system catalogs to estimate sizes & costs
  - Considers combination of CPU and I/O costs.
  - System R's scheme has been improved since that.
- **Search Algorithm**

Selectivity Estimation

Selectivity of an operator: an approximation for what percentage of pages will make it through the operator onto the operator above it. selectivity = |output| / |input|.

- **X=a**: 1/(unique vals in X)
- **X=Y**: 1/max(unique vals in X, unique vals in Y)
- **X>a**: (max(X) - a) / (max(X) - min(X) + 1)
- **cond1 AND cond2**: Selectivity(cond1)*Selectivity(cond2)
- Selectivity of **Joins**:
  - if no condition: |A|*|B|.
  - With condition A.id=B.id: |A|*|B| / max(unique vals for A.id, unique vals for B.id)
  - Recall from algebraic equivalences

    $R \bowtie_p \sigma_q(S) \equiv \sigma_p(R \times \sigma_q(S)) \equiv \sigma_{p \wedge q}(R \times S))$
    - Hence selectivity just $s_p s_q$
      - Applied to |R| × |S|!
  - Total rows: $s_p s_q$|R||S|

(capital letters are for columns and lowercase letters represent constants)

Common Heuristics for Plan Space

1. Push down projects (π) and selects (σ) as far as they can go, e.g:
   - $\pi_{sname}(\sigma_{(bid=100 \wedge rating > 5)}(Reserves \bowtie_{sid=sid} Sailors))$
   - $\pi_{sname}(\sigma_{bid=100}(Reserves) \bowtie_{sid=sid} \sigma_{rating > 5}(Sailors))$
   - $\pi_{sname}\sigma_{(bid=100 \wedge rating > 5)}(Reserves \bowtie_{sid=sid} Sailors)$
   - $\pi_{sname}(\pi_{sid}(\sigma_{bid=100}(Reserves)) \bowtie_{sid=sid} \pi_{sname, sid}(\sigma_{rating > 5}(Sailors)))$
2. Only consider left deep plans
3. Do not consider cross joins unless they are the only option (Avoid Cartesian products)

Upshot

- Compute **selectivities** for basic predicates: The original Selinger version; The histogram version.
- **Assumption** 1: uniform distribution within histogram bins: Within a bin, fraction of range = fraction of count
- **Assumption** 2: independent predicates
  - Selectivity of AND = product of selectivities of predicates
  - Selectivity of OR = sum of selectivities of predicates - product of selectivities of predicates
  - Selectivity of NOT = 1 – selectivity of predicates
- **Joins** are not a special case: Simply compute the selectivity of all predicates, and multiply by the product of the table sizes.

Transactions & Concurrency

Transactions

A transaction ends in 2 ways: it either **commits** or **aborts**.

ACID: High-Level Properties of Transactions

- Atomicity: All actions in the Xact happen, or none happen.
- Consistency: If the DB starts out consistent, it ends up consistent at the end of the Xact
- Isolation: Execution of each Xact is isolated from others.
- Durability: If a Xact commits, its effects persist.

Serial schedule: each transaction runs from start to finish without any intervening actions from other transactions.

Serializable: S is equivalent to some serial schedule.

Equivalent: 2 schedules must:

- involve the same transactions
- each individual Xact's actions are ordered the same
- both schedules leave the DB in the same final state

Conflicting operations:

- The operations are from different transactions
- Both operations operate on the same resource
- At least one operation is a write

Conflict equivalent: Two schedules involve the same actions of the same transactions, and every pair of conflicting actions is ordered the same way.

- A stronger condition than being equivalent.

Conflict serializable: Schedule S is conflict equivalent to some serial schedule (Implies S is also Serializable). S is conflict serializable iff its dependency graph is acyclic.

Dependency Graph: One node per Xact. Edge from Ti to Tj if: An operation Oi of Ti conflicts with an operation Oj of Tj and Oi appears earlier in the schedule than Oj.

Lock

Two Phase Locking (2PL) Rules: Xact must obtain a S (shared) lock before reading, and an X (exclusive) lock before writing. Xact cannot get new locks after releasing any locks. 2PL guarantees conflict serializability, but, does not prevent cascading aborts.

Strict 2PL: 2PL + All locks released together when Xact completes: Xact has committed (all writes durable), OR Xact has aborted (all writes have been undone)

Lock Management: handles lock and unlock requests. Maintains a hash table, keyed on names of objects being locked. keeps an entry for each currently held lock.

Entry contains:

- Granted set: Set of Xacts currently granted access to the lock
- Lock mode: Type of lock held (shared or exclusive)
- Wait Queue: Queue of lock requests

	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T3(X) ← T4(X)
B	{T6}	X	T5(X) ← T7(S)

When lock request arrives: Does any Xact in Granted Set or Wait Queue want a conflicting lock?

- If no, put the requester into "granted set" and let them proceed
- If yes, put requester into wait queue (typically FIFO)

Deadlock: Cycle of Xacts waiting for locks to be released by each other. 3 ways of dealing with deadlocks:

- Prevention
- Avoidance
- Detection and Resolution

Deadlock Detection: Create and maintain a "waits-for" graph, periodically check for cycles in a graph.

Multiple Granularity Locking Protocol

- Each Xact starts from the root of the hierarchy.
- 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.
- Protocol is correct in that it is equivalent to directly setting locks at leaf levels of the hierarchy.

Recovery

Buffer Management Plays a Key Role.
Recovery Manager guarantees Atomicity & Durability.

Steal, No force

Force policy: when a Xact finishes, force all modified data pages to disk before the Xact commits.
Ensure durability because disk is persistent, once a page makes it to disk it is saved permanently. The downside of this approach is performance.

No force policy: only write back to disk when the page needs to be evicted from the buffer pool.
Reduce unnecessary writes. But complicates durability because if the database crashes after the transaction commits, some pages may not have been written to disk and are consequently lost from memory, since memory is volatile. We will redo certain operations during recovery.

No-steal policy: pages cannot be evicted from memory (and thus written to disk) until the transaction commits.
Ensures that we do not leave the database in an intermediate state because if the transaction does not finish, then none of its changes are actually written to disk and saved. Problem: Have to keep every modified page in memory until a transaction completes.

Steal policy: allows modified pages to be written to disk before a transaction finishes.
This will complicate enforcing atomicity, fix this problem by undoing bad operations during recovery.

Simple idea: No-steal, Force (didn't achieve atomicity)

Our Policy: Steal, No-Force.

Log Records

log: a sequence of log records describe the operations that the DB has done. With a write buffer ("tail") in RAM.

UPDATE Log Record
<XID, pageID , offset , length , old data , new data>

- XID: which transaction did this operation
- pageID: what page has been modified
- offset: where on the page the data started changing (typically in bytes)
- length: how much data was changed (typically in bytes)
- old data: what the data was originally (used for undo operations)
- new data: what the data has been updated to (used for redo operations)

Other Log Records
<LSN, prevLSN, XID, type>

COMMIT: signifies a Xact is starting the commit process.

ABORT: signifies a Xact is starting the aborting process.

END: signifies that a Xact is finished (usually means that it finished committing or aborting).

Write-Ahead Logging (WAL)

Use WAL to allow STEAL/NO-FORCE without sacrificing correctness.

Write-Ahead Logging Protocol:

1. Log records must be written to disk **before** the corresponding data page gets written to disk. Achieve **atomicity**.

The intuition for this is that if a data page is written first and then the database crashes we have no way of undoing the operation because we don't know what operation happened!

2. All log records must be written to disk **before** a Xact commits. Achieve **durability**.

The intuition is that we need to persistently track what operations a committed transaction has performed. Otherwise, we would have no idea what operations we need to redo. By writing all the logs to disk, we know exactly which operations we need to redo in the event that the database crashes before the modified data pages are written to disk!

LSN (Log Sequence Number): Each log record has a LSN.
LSN is a unique increasing number that helps signify the order of the operations. LSNs identify log records; linked into backwards chains per transaction (via prevLSN).

prevLSN: stores the last operation from the same Xact. the previous log record written by this XID.
prevLSN will be useful for undoing a transaction.-> records of an Xact form a linked list backwards in time.

flushedLSN: keeps track of the LSN of last log record that has been flushed to disk. tracked in RAM.
When a page is flushed, it means that the page has been written to disk; it usually also implies that we evict the page from memory because we don't need it there anymore. The flushedLSN tells us that any log records before it should not be written to disk because they are already there.

pageLSN: stores the LSN of the operation that last modified the page. Each data page in DB contains one.
Tell us what operations actually made it to disk and what operations must be redone.
pageLSN allows comparison of data page and log records.

Aborting a Transaction: 1. write an ABORT record to the log to signify that we are starting the process. 2. start at the last operation in the log for that Xact. 3. undo each operation in the Xact and write a CLR record to the log for each undone operation.

CLR (Compensation Log Record): record signifying that we are undoing a specific operation.
It is essentially the same thing as an UPDATE record (it stores the previous state and the new state), but it tells us that this write operation happened due to an abort.

Recovery Data Structures

We will keep **Transaction table** and **Dirty Page Table** to make the recovery process a little bit easier. Both of these tables are stored in memory; when recovering from crash, you will have to use the log to reconstruct the tables.

Transaction table: stores information on t he active Xacts.

- XID: transaction ID
- status: either running, committing, or aborting
- **lastLSN**: the LSN of the most recent operation for this transaction

Dirty Page Table (DPT): keeps track of what pages are dirty. This information tell us what pages have operations that have not yet made it to disk.

- Page ID
- **recLSN**: the first operation to dirty the page

Inequality Questions

- Before page i is allowed to be flushed to disk, what inequality must hold? ***pageLSN_i <= flushedLSN***

We must flush the corresponding log records before we can flush the data page to disk. A data page is only flushed to disk if the LSN of the last operation to modify it is less than or equal to the flushedLSN.

- All the logs must be flushed to disk before a transaction T can commit: ***flushedLSN >= lastLSN_T***

If the flushedLSN is is greater than the last operation of the Xact then we know all of the logs for that Xact are on disk.

- For a page P in the DPT, the following inequality must always be true: ***recLSN_P** <= in memory **pageLSN_P**, **recLSN_P** > on disk **pageLSNP***.

If the page is dirty, then the operation that dirtied the page (recLSN) must not have made it to disk, so it must have came after the operation that make it to disk for that page.

ARIES Recovery Algorithm

- Start from a checkpoint (found via master record)
- Analysis Phase: reconstructs the Xact Table and the DPT
- Redo Phase: repeats operations to ensure durability
- Undo Phase: undoes operations from transactions that were running during the crash to ensure atomicity

Possible log record types:

- Update, Commit, Abort
- Checkpoint (for log maintenance)
- Compensation Log Records (CLRs) (for UNDO actions)
- End (end of commit or abort)

Checkpointing: writes the contents of the Xact Table and the DPT to the log.

- begin_checkpoint record: Indicates when ckpt began.
- end_checkpoint record: Contains current Xact table DPT.

Analysis Phase
Purpose: to rebuild what the Xact Table and the DPT looked like at the time of the crash.
To do this, we scan through all of the records in the log beginning from the record immediately after the begin checkpoint record. We modify our tables according to the following rules:

- On any record that is not an END record: add the transaction to the the Xact Table (if necessary). Set the lastLSN of the transaction to the LSN of the record you are on
- If the record is a COMMIT or an ABORT record, change the status of the transaction in the Xact Table accordingly
- 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
- If the record is an END record, remove the transaction from the Xact Table.

Redo Phase
Redo Phase ensures durability. We will repeat history in order to reconstruct the state at the crash.
We start at the smallest recLSN in the DPT because that is the first operation that may not have made it to disk. We will redo all **UPDATE** and **CLR** operations **unless** one of the following conditions is met:

- The page is not in the DPT. If the page is not in the DPT it implies that all changes (and thus this one!) have already been flushed to disk.
- recLSN > LSN. This is because the first update that dirtied the page occurred after this operation. This implies that the operation we are currently at has already made it to disk, otherwise it would be the recLSN.
- pageLSN (on disk) ≥ 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.

Undo Phase
Undo phase ensures atomicity.
The undo phase will start at the end of the log and works its way towards the start of the log. It undoes every UPDATE (only UPDATEs!) for each transaction that was **active (either running or aborting)** at the time of the crash so that we do not leave the database in an intermediate state. It will not undo an UPDATE if it has already been undone (i.e. a CLR record is already in the log for that UPDATE).
For every UPDATE the undo phase undoes, it will write a corresponding CLR record to the log. CLR records have one additional field that we have not yet introduced called the **undoNextLSN**. The undoNextLSN stores the LSN of the next operation to be undone for that transaction (it comes from the prevLSN of the operation that you are undoing). Once you have undone all the operations for a transaction, write the END record for that transaction to the log.

ER Modeling

Entity & Relationship: 2 main components

- **Entity**: a real-world object described by a set of attribute values. An entity is usually depicted as a rectangle.

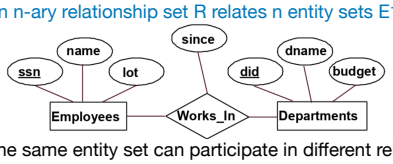
Attributes/fields: the attributes/fields associated with the entity are depicted as ovals. Some attributes is underlined because it is the identifying attribute for this entity.

- **Relationship**: association among two or more entities. A relationship is usually depicted as a diamond.

Again, the **attributes** associated with the relationship are depicted as ovals.

Entity Set: A collection of similar entities.
Each entity set has a key (underlined).

Relationship Set: Collection of similar relationships.
An n-ary relationship set R relates n entity sets E1 ... En.



The same entity set can participate in different relationship sets, or in different "roles" in the same relationship set.

Relationship Constraints

many-to-many relationship: each entity can participate in 0 or more times in the relationship and vice versa. thin black line.

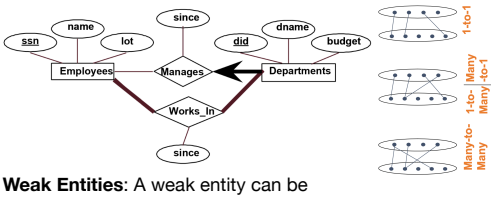
Key constraints—denotes a **1-to-many** relationship: at most one. thin arrow.

Participation constraints—denotes an **at least one** relationship: at least one. thick line.

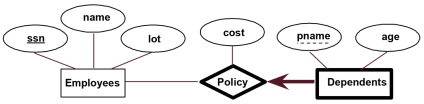
- **total** constraint: thick.
- **partial** constraint: thin.

理解：constraints定义的是entity与relationship之间的关系，作用的对象可以理解为与relationship->没有什么方向上的对称可言。

For example, in the following figure, we added two thick lines. One of them denotes that each employee works in at least one department. The other denotes that each department has at least one employee. In addition, we added a thick arrow pointing from departments to manages. This is both a key and participation constraint: each department has at least and at most one manager.



Weak Entities: A weak entity can be identified uniquely only by considering the primary key of another (owner) entity.
Owner entity set and weak entity set must participate in a 1-to-many relationship set (1 owner, many weak entities).
Weak entity set must have total participation in this identifying relationship set.
We depict a weak entity by bolding the rectangle and relationship. Weak entities have only a "partial key" (dashed underline).
In the example below, we say that every dependent is uniquely identified by their pname and an employees' ssn and that each employee has 0 or more dependents attached to them.



Data Warehouse/Lake

Data Warehouse/Lake

Data warehouse: collects and organizes historical data from multiple sources.
A **data lake** is a vast pool of raw data, the purpose for which is not yet defined. A **data warehouse** is a repository for structured, filtered data that has already been processed for a specific purpose.

Data Lake: 存储原始数据（未加工处理的数据，结构化和非结构化均可）的池子，后续根据具体的用途使用时加工处理；**Data Warehouse**: 存储经过加工处理结构化的数据的仓库；

Data is periodically **ETL**ed into the data warehouse:

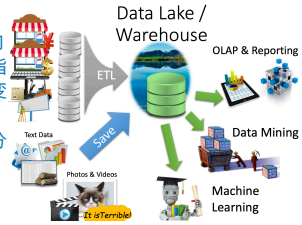
- **Extracted** from remote sources
- **Transformed** to standard schemas
- **Loaded** into the (typically) relational system

Online Transaction Processing (**OLTP**) v.s. Online Analytics

Processing (**OLAP**)

OLTP 系统强调数据库内存效率，强调内存各种指标的命令率，强调绑定变量，强调并发操作；

OLAP 系统则强调数据分析，强调SQL执行市场，强调磁盘I/O，强调分区等。



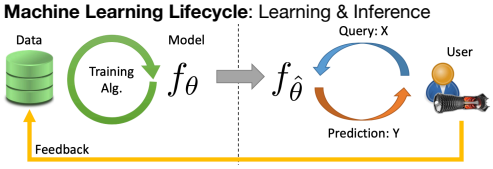
Knowledge Discovery in Databases (KDD)

Process of extracting knowledge from data.
Descriptive statistic -> Inferential Statistic.

- **Data Selection**: What data do I need for a given task?
- **Data Cleaning**: Preparing the data for a given task
- **Data Mining & ML**: Running algorithms to infer patterns
- **Evaluation**: Verifying that patterns are significant

Learning

Machine Learning Lifecycle: Learning & Inference



Typically a **time consuming** iterative batch process:

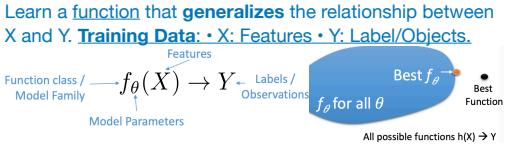
- Feature engineering
- Validation

Focus is on **making fast robust predictions**:

- Monitoring and tracking feedback
- Materialization + fast model inference

Fitting the Model

Learn a function that **generalizes** the relationship between X and Y. **Training Data**: • X: Features • Y: Label/Objects.



Clustering

K-means

```
centers ← pick k initial Centers
while (centers are changing) {
  // Compute the assignments (E-Step)
  asg ← [(x, nearest(centers, x)) for x in data]
  // Compute the new centers (M-Step)
  for i in range(k):
    centers[i] =
      mean([x for (x, c) in asg if c == i])
}
```

Compute the "Expected" Assignment

Find centers that maximize the data "likelihood"

Guaranteed to converge! … to what? To a **local optimum.** ☹ Depends on Initial Centers

Picking the Initial Centers: K-means++

- Randomized approximate algorithm
- Intuition select points that are not near existing centers

Steps:

(1) Suppose I pick this center first
(2) Pick the next center not near the previous center
(3) Pick the final center not near previous two centers

K-Means in Map-Reduce

- **MapFunction**(old_centers, x)
  - Compute the index of the nearest old center
  - Return (key = nearest_centers, value = (x, 1))
- **ReduceFunction** combines values and counts
  - For each cluster center (Group By)
- Data system returns aggregate statistics:

$$s_i = \sum_{x \in Cluster\ i} x_i \text{ and } n_i = \sum_{x \in Cluster\ i} 1$$

- ML algorithm computes new centers: $\mu_i = s_i / n_i$

Interacting With the Data

Compute Locally

Request data sample, Get sample of data.
Good for smaller datasets: Faster more natural interaction; Lots of tools!

Cluster Compute

Query: f_θ, Response: $\sum$
Good for bigger datasets and compute intensive tasks.

Regression

Least Squares

Squared error comes from Gaussian noise assumption.

$$Y \sim N(\theta^T x, \sigma^2)\ p(y|x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y-\theta^T x)^2}{2\sigma^2}\right)$$

Minimizing the Squared Error

$$\hat\theta = \arg\min_{\theta \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} (y_i - \theta^T x_i)^2$$

Statistical Query Pattern

$$\hat\theta = (X^T X)^{-1} X^T Y$$

Test Error = Noise + Bias2 + Variance
Model Complexity increase: variance+, bias-.

Closed form solution for **Ridge** regression (L2):

$$\hat\theta_{ridge} = (X^T X + \lambda I)^{-1} X^T Y$$

HW & Quiz & Other

- In an **equidepth** histogram, if you have an entry with low frequency, then it's either going to get clumped together with a bunch of other low frequency entries or with 1 other high frequency entry. So An equidepth histogram gives worse resolution on low frequency entries than a **equiwidth** histogram
- Sort-Merge Join, Clustered Index Traversal will result in an interesting order in a query where we require the output to be sorted
- Larger **reduction factor** -> higher |output|/|input| ratio
- **ARIES** we covered uses strict 2PL, and system does not use strict 2PL cannot be using ARIES.
- 单射（**injection**）每一个x都有唯一的y与之对应
- 满射（**surjection**）每一个y都必有至少一个x与之对应
- 双射（又叫一一对应，**bijection**）每一个x都有y与之对应，每一个y都有x与之对应
- A **weak entity** behaves exactly like an entity with a key constraint and total participation. -> False. Weak entities cannot be uniquely identified w/o their owner identities.
- the **Reservoir Sampling Algorithm** runtime: O(N).
- **Regularization** is a way of mitigating overfitting. Increasing the regularization parameter will increase bias, decreasing the regularization parameter (overfitting) will increase variance.

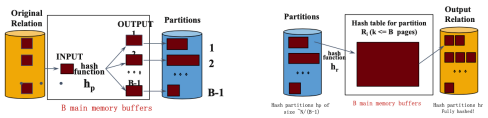
Appendix

1. LSN list

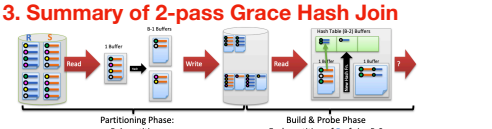
LSN: stored in each log record. Unique, increasing, ordered identifier for each log record.
flushedLSN: stored in memory, keeps track of the most recent log record written to disk.
pageLSN: LSN of the last operation to update the page (in memory page may have a different pageLSN than the on disk page).
prevLSN: stored in each log record, the LSN of the previous record written by the current record's transaction.
lastLSN: stored in the Xact Table, the LSN of the most recent log record written by the transaction.
recLSN: stored in the DPT, the log record that first dirtied the page since the last checkpoint.
undoNextLSN: stored in CLR records, the LSN of the next operation we need to undo for the current record's Xact.

2. Two Phases of Hash

- Partition: (Divide)
- Rehash: (Conquer)



3. Summary of 2-pass Grace Hash Join



- Partitioning phase: read+write both relations
  - 2([R]+[S]) I/Os
- Matching phase: read both relations, forward output
  - [R]+[S]
- Total cost of 2-pass hash join = 3([R]+[S])
- **Memory Requirements?**
- Build hash table on R with uniform partitioning
  - **Partitioning Phase divides R into (B-1) runs of size [R] / (B-1)**
  - **Matching Phase requires each [R] / (B-1) < (B-2)**
  - **R < (B-1) (B-2) = B²**
- Note: no constraint on size of S (probing relation)!

4. Multidimensional Data Model

Sales Fact Table

pid	timeid	locid	sales
11	1	1	25
11	2	1	8
11	3	1	15
12	1	1	30
12	2	1	20
12	3	1	50
12	1	1	8
13	2	1	10
13	3	1	10
11	1	2	35
11	2	2	22
11	3	2	10
12	1	2	26

Locations

locid	city	state	country
1	Omaha	Nebraska	USA
2	Seoul		Korea
5	Richmond	Virginia	USA

Products

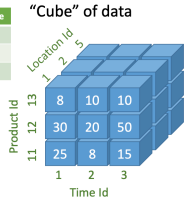
pid	pname	category	price
11	Corn	Food	25
12	Galaxy 1	Phones	18
13	Peanuts	Food	2

Time

timeid	Date	Day
1	3/30/16	Wed.
2	3/31/16	Thu.
3	4/1/16	Fri.

Dimension Tables

- Multidimensional "Cube" of data



5. Least Squares Regression using the Query Pattern Pattern

$$\hat\theta = (X^T X)^{-1} X^T Y$$

➢In database compute sums:

$$C = X^T X = \sum_{i=1}^{n} x_i x_i^T \qquad O(np^2)$$

$$d = X^T y = \sum_{i=1}^{n} x_i y_i \qquad O(np)$$

➢On client compute:

$$\hat\theta = C^{-1} d \qquad O(p^3)$$

6. Gradient Descent Algorithm

```
t ← 0
θ(0) ← Vec(0)
while (not converged):
  θ(t+1) ← θ(t) - η * G(θ; X,y)
  t ← t + 1
```

➢Does this fit the statistical query pattern

- Yes! Only dependence on data is:

Size? $G(\theta; X, y) = \frac{1}{n} \sum_{i=1}^{n} (y - \theta^T x_i) x_i$ Complexity? $O(np)$

7. Regularization to Reduce Over-fitting

➢We can add a regularization term:

$$\hat\theta = \arg\min_{\theta \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} \underbrace{(y_i - \theta^T x_i)^2}_{\text{Squared Loss}} + \underbrace{\lambda}_{\text{Regularization Parameter}} \underbrace{R(\theta)}_{\text{Regularization Function}}$$

➢Common of Regularization Functions:

Ridge (L2-Reg) Regression $R_{\text{Ridge}}(\theta) = \sum_{i=1}^{d} \theta_i^2$

Lasso (L1-Reg) $R_{\text{Lasso}}(\theta) = \sum_{i=1}^{d} |\theta_i|$

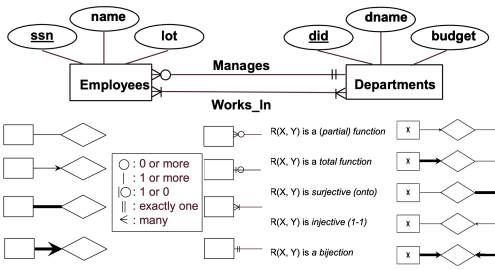
- Encourage small parameter values

➢The parameter λ determines amount of reg.

- Larger → more reg. → lower variance → higher bias

8. Crow's Foot Notation

No relationship attributes



9. Lock Compatibility Matrix

	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