Notation: I am using ∞ for the weird relational infinity sign but with rigid triangles, and using TXN for transaction.

Relational Algebra

Base Operators

- **Select** (in sql this is where, row conditional)
  Implementation choices (query optimizer tries to pick what's more eff.):
  Index: If B+ tree already exists just use that to range search, would save time if index is on the conditional. However, if the table is small it could be slower to use index than just scan the entire table
  Scan: Typically cheapest option w/o index, cheap bc sequential reads to disk/db is ok.
- **Project** (in sql this is select, pull out columns)
  Because when doing a projection it might make the output tuples contain duplicates (ex: SELECT name from employees), and not set (breaking our class assumptions) so we have to consider that
  Implementation (of ensuring distinct): We could build a hashtable to contain all of these values, but the hash table might get too large in memory causing thrashing.
- **Cartesian Product** (takes 2 tables, pairs together each tuple combo)
  Super super expensive. Equivalent of T1,T2. (ex: 100K tuples in 2 tables being joined output 10M tuples). In practice rarely executed because there are more optimized ways to join a table.
- **Union**
  Returns all tuples in R1 or R2, must have the same schema. We assume output is a **set** for this class aka distinct
  Implementation: Use a hashtable or sort
- **Set Difference**
  All tuples in R1 not in R2 (R1-R2). Requires the same schema.
  Implementation: use HashTable, build hash on R2. Iterate on R1 if tuple's hash in R2 ignores it else emit tuple. We do not want to hash R1 because if we did this we'd need to loop over R2 then if the t2 tuple is in R1's hash table we'd need to remove it from R1 then at the end loop over R1 again and emit the remaining tuples. In the real world the hash tables usually have a B+ tree like structure so deletions can be expensive, and this gets weird with duplicates (which aren't considered in this class.

Derived Operators

- **Intersection**
  $R1 \cap R2 \equiv R1 - (R1 - R2)$ inefficient
  Implementation: Hash T1 (opt: in an index) scan T2 for each row hash if in T1 emit row.
  We assume that the hash fits into memory and lives in the buffer somewhere. We read each tuple in T1 then probe it against the hash table in memory 0 cost.
- **Join**
  Most painful to implement and costly. Can be derived from selection over cartesian product, but this is dumb.
  Join types
  Theta join - arbitrary condition $R1 \infty_{\theta} R2$
  Natural join - join on the shared columns names when they are equal. Renames the duplicate cols to like col1_x col1_y on this join
  Equi Join - Join table under condition, 2 attributes are equal in value (not name) natural join is a particular case of this join

Join Implementation

Most difficult thing to implement this fast, assumptions 2 tables R,S and we're doing an equi join $R \infty_{x=y} S$

Implementation types

Nested Loop Join (BNL)

For t1 in R: for t2 in S: if cond emit. But we can be more efficient at page level, we can read in as much from R as possible into memory, leaving 1 page for S and 1 page for output buffer. Reduces our **cost to** $|R| + ceil(\frac{|R|}{B-2}) x |S|$

Note: We **always** want the smaller table to be the inner table to be smaller |S| In this equation, on exams when in doubt look at it both ways to see what's more efficient. Idk TA also said: We can assume that the small table is the outer table in the exam.

Sort Merge Join

Sort each table using external sort, use 2 ptrs to read in the tables and for each tuple in the outer table scan the inner table by moving the ptr. When we move the outer pointer increased by 1 we have to reset the inner ptr, because if inner tuples are all the same value on our cond we need to emit those too.

Best Merge Case: $|R| + |S|$ Each page is distinct, only need to read each page once. **Worst Merge Case:** $|R| * |S|$ if all values the same will need to output combo of all

Total Cost (if $M \le B^2 \& N \le B^2$): 5M+5N, **Worst Case:** $4M + 4N + (M x N)$

Hash Join

Hash both R and S into N buckets based on the condition columns (not good for range). Once the tables are split up into buckets, they still need to be merged but since they're smaller tables they can be merged quicker and don't need to look across buckets because hash (basically map reduce). This works under the **assumptions**: R1 fits into memory $|R_1| \le B - 2$ and we assume uniform distribution across buckets.

Total Cost: 3|R| + 3|S| -> hash cost is 2|R|+2|S| and the merge is |R|+|S|, **note:** this only works if the assumptions above hold, otherwise cost increases

Index Nested Loop Join

If we have an index on the condition that's being joined, we can use that. for r in R: for tuple in S where r_i=s_j (we can do this eff. w/ tuple): add <r,s> to result.

Cost: $|R| + (|R| * |S| * cost\ of\ probing\ S's\ index\ to\ find\ tuples)$, note: avg cost of probing index is 1.2 for hash index and 2-4 for B+ tree

Query Plan & Relational Algebra

Ex: Select N from E where a > 30. Query Plan is (Select N <- $\theta_{a>30}$ <- E) as relational expression $\pi_n(\sigma_{a>30} E)$

Relational Algebra has Limitations

- Cannot compute transitive closure. Ex: we have a table of relations, name1 name2 relation: (Fred, Mary, Father), (Mary, Joe, Cousin), (Mary, Bill, Spouse), (Nancy, Lou, Sister). We want to compute all relatives to Fred which we want to return Mary, Joe, Bill, but we used transitive closure to find non-explicitly mentioned people which is not supported in relational algebra since it operates on a single tuple.

Sorting

- **In memory sorts**: quicksort O(n log n) assumes it's in memory which is O(1) for random access. Will thrash if not enough pages in memory and will be slow :(the random access pattern is bad for disk.
- **Disk Resident Sort/External Sort**
  With just 3 memory pages we can sort fast. We can use this sort for joining tables under a condition instead of a hash and other things.
  1. Sort each page independently with quick sort, and write sorted page back to disk (only requires 1 page at a time)
  2. Read the first pages together and merge them into a longer page. (requires 3 pages: 1 page per group and 1 flush page)
     When an accessing page runs longer than 1 page we can use a 2 ptr comparison since each page is already sorted.

Optimally we want to use as many memory pages as possible, since we only need 1 page to flush to disk if we had 20 pages in memory we can use 19 pages each for their own run so we can merge 19 runs at once reducing the # of iterations we need.

Generalized cost: N pages, B buffer pages to use in the sort. # of runs = $1 + ceil(\frac{N}{B})$ -> total cost $2N * (1 + ceil(log_{B-1}(ceil(N/B))))$ (we need 1 read 1 write per level)

Typical Case: B buffer pages, M file pages if $M < B^2$ then cost of sort is $4M$, (or $M \le B(B-1)$, the B^2 is just what we use bc it's easier)
Because pass 0: create runs of B pages long, pass 1: create runs of B(B-1) pages long (if M<B(B-1)) we are done

Query Optimization

Query Parser takes in SQL outputs a Plan P (logical plan), optimizer tries to find a good enough plan in a quick amount of time to execute. Optimizer picks the physical execution plan, which has details about exactly what to do, like what type of joins to use (ex: index, hash, sort, etc). Optimizer calculates plan costs to estimate how good a plan is.

On the fly (pipelining) - Pull tuple into memory, if cond not met discard otherwise pipeline through (no intermediary writes to disk)

Ex: $\pi_{sname}\ (on\ the\ fly)\ <-\ \sigma_{bid=100\ \&\ rating=100}\ (on\ the\ fly)\ <-\ Results \infty_{sid=sid} Sailors\ (simple\ nested\ join)$ -> cost is just the cost of joining bc rest is done on the fly in memory.

We can push the selection before the join if we wanted to, this will reduce the size of the table we need to join, however we lose the ability to do it on the fly, because the intermediary tables need to be written to disk before they're joined. We also need to make a lot of assumptions about if this is a better cost plan because we don't have time to evaluate the exact sizes of these tables after the selection to see if it's a quicker plan than just doing the selection after. Ex: cost estimation assuming 5 buffer pages. Scan reserves table (size 1000 pages, given) + write to temp T1 (10 pages if 100 boats uniformly distributed, assumption). Scan sailors (size 500 pages given) write temp T2 (250 pages if we have a uniform distribution of ratings across 10 discrete values). Sort T1: 4*10 (because $10 \le 5^2$). Sort T2: 4*2*250 (external sort general formula), Merge: 10+250. Total: 4060.

If instead of sorting we used nested join: 10 + 4*250 = 2770

If we push projections earlier we can condense tables even smaller. If this causes T1 to fit into 3 pages BNL which drops cost to below 2000.

Push Projection/Selection Downside: We might accidentally destroy indexes since we're creating a new table which doesn't have an index on it anymore, so we have to be careful.

How to estimate cost

Use search algorithms to quickly find plans, use A* algorithm to find good plans with some heuristic. Since brute force will not work well bc we only have a few ms to make & start a decision.

We need to rely on a ton of assumptions about the tables. Some naive assumptions are like uniform assumptions, assuming how many tuples after a condition is applied is difficult.

One approach: as part of maintenance mode like at 2am we update statistics about a table like variance, mean, etc. Can't compute these during query optimization bc it'll take too long. Idea is these statistics might change as new rows added/deleted since the change but will be good enough to make assumptions that they're correct. **For the final we should be stating and making a lot of uniform distribution assumptions to estimate cost**

Multi-Table Joins

Left Deep Join

This is the type of multi table join that's used because it's like $((A \infty B) \infty C) \infty D$ depending on the join type of the operator used the left/outer table might not need to be on disk so we save intermediary writes, and can just pipeline the results to the rest of the joins saving on IO ops. On the right side of the joins they need to be on disk so we'd have extra writes to write intermediary values.

Also, when indexes are present we make better use of them because if we're writing intermediary results we cannot use those indexes anymore, so we're retaining indexes on the inner tables for more eff lookup when pipelining. When calculating cost, I think we just assume we can pipeline the outer/left tables through the rest of the operations so we don't count the writes out.

Right Deep Join (worse bc have to write the big joined table to disk as an intermediary step)

Bushy Joins (still has to write intermediary values to disk)

Generating Physical Plans - many different ways, can permute all options maybe with some probability, can use heuristics to know where costs are likely to come from, and use domain knowledge/heuristics to make good ones

Transaction Management / Concurrency Control

Transactions - Must be executed atomically, if fails or crashes it must rollback any changes made by the transaction.

ACID - Atomic (transactions executed atomically), Consistency (user defined, but we assume before and after a transaction is consistent), Isolation (2 transactions running at same time can't interfere with each other), durable (changes persist to disk when transaction complete)

Atomic - We will use locks to make sure we are atomic, but to be efficient we need to look at the smallest known attribute (usually a row/column) that might get changed because if we locked the whole table that stops a lot of throughput. We can have 10K-20K transactions executing at once and only a handful will need to use the same locks. Sometimes a deadlock occurs :(

Consistency - User's responsibility to define what a "consistent" transaction is that maintains it.

Durable - persisted to disk & cannot be lost.

ACID is great, but it makes databases slow-ish especially at scale. For big data ACID is too slow and so other databases relax some constraints of ACID such that it can be faster (ex: NoSQL, eventual consistency, etc)

Transaction / Crash Recovery

We will consider the case of a system crash (ex: power goes out). We use a log to record every action of every transaction. In adhoc SQL like when writing SQL in the psql cli it just automatically creates a transaction, but when using embedded sql like a driver in some other language like Python you can manually define when a transaction starts and ends, and also rollback/cancel a transaction.

We assume that a transaction reads and writes some elements (types of elements: cells, tuple, table, etc)

Primitive Ops of TXN:
Input(X) - read x to buffer
Read(X,t) - copy X to txn local variable t
Write(X,t) - Copy TXN local var t to element X
Output(X) - Write X to disk

The Log
We will use an append only file containing TXN logs/records, because appending is fast. Multiple TXN run concurrently so log records are interleaved. If a system crashes, we will restore the database based on what we know from the log.

Log Keywords/Operations
<Start T> - TXN T has begun
<T,X,v> - T updated X and **old value** was v
<Commit T>
<Abort T>

Undo Logging Rules
R1: If TXN T modifies X, then <T,X,v> **must be written to disk** before **X** is written to disk.
R2: If TXN T commits, then <Commit t> must be written to disk **only after** all changes T makes are written to disk.

Checkpointing
Periodic Checkpointing: Pause database from accepting new transactions and wait until current ones finish then write <CKPT> to log. We only need to scan up until this.
Nonquiescent Checkpointing: We want to allow more transactions to come in while checkpointing. To do this, we will write <START CKPT([list of all currently running TXNs])>, but we continue to accept more transactions, when the transactions that were running when we started the checkpoint finish we write <END CKPT>. When doing crash recovery when we run into <END CKPT> we now only need to scan up until <START CKPT>, because at the start CKPT had all running TXNs at the time.
Redo Logging - We didn't cover much in class, just briefly mentioned that instead of storing the old values in the log we store the new values and do a top down log recovery approach.

Normalization

Schema Normalization - The process of elimination (or reducing) data redundancy. (ER -> Convert to Schema -> List all FDs (ask business ppl))
Redundancy - Repetition of data
Even if we convert an ER diagram correctly, and there's no redundancy from that we can still introduce redundancy with constraints on our table, for example we can have functional dependency.
Functional Dependency - FD f: x->y. Set X determines Y. If T1,T2 both have an X1 they share, they must agree on y
Example: If an employee's job title determines their phone # (maybe phone # is department wide)

Forms of Schema
1st Normal Form - Each attribute is atomic (no arrays/sets).
2nd Normal Form (AnHai said: old & obscure) - Needs to be in 1NF, plus partial Dependency, meaning that non-primary attributes are fully functionally dependent on the primary key. For example, in a table where a composite key exists (like EmployeeID and ProjectID), all other attributes should depend on both keys, not just one of them.
3rd Normal Form - Needs to be in 2NF, plus it has no Transitive Dependency, meaning that non-primary key columns should not depend on other non-primary key columns. In simpler terms, each non-primary key attribute must directly depend on the primary key.
Boyce Codd Normal Form (BCNF) - Need to be in 3NF, plus for every functional dependency (X → Y), X should be a super key. This means every determinant must be a candidate key. BCNF is a stricter version of 3NF where no non-trivial functional dependencies of attributes on anything other than a candidate key are allowed.
4th Normal Form - Needs to be in BCNF, plus it has no Multivalued Dependencies, other than trivial ones. This means that for every one of its non-trivial multivalued dependencies X →→ Y, X is a superkey.
Downside: We pay for less data redundancy bc we need to do a lot more joins which causes query performance to suffer. We combat this by joining tables that are often joined by queries **together in advance** with **views** and save it to disk.

Example Problems

What is the difference between logical and physical operators? Give an example of each.
A: A logical operator specifies the type, such as join, selection, projection, etc. But it does not specify the exact implementation to be used. A physical operator specifies the implementation, such as a block nested loop join, a sort-merge join, etc. A logical operator can be associated with multiple possible physical operators.

Cost Calculation: Given two relations R and S with size B(R) = 10000 and B(S) = 8000:
a. Suppose the amount of memory available is M = 2001. Compute the disk IO cost of a nested-loop join for R and S.
A: The smaller table must be the outer table, which is table S in this case. We will need ceiling[|S|/(M-2)] iterations, in each iteration we need to read through the entire table R. So the total cost of reading R is ceiling[|S|/(M-2)] * |R|. We also read through S once, and the cost of that is |S|. We are ignoring the cost of writing the output. So the total cost is ceiling[|S|/(M-2)] * |R| + |S|.
b. Suppose the amount of memory available is M=6000. Is it possible to perform a nested-loop join for R and S with cost no more than 25000 disk IO? Explain your answer.
A: the correct formula is [|S|/(M-2)] * |R| + |S| = 28000. It is not possible to join for less than 25000.

SQL -> Logical Query Plan: Given Company(cname, city, president), Product(pname, maker, price). Assume B(Company) = B(Product) = 10000 blocks. Assume further that each block can accommodate 5 records, and that the memory has 101 buffers. Consider Query: SELECT Company.cname, Company.president FROM Company, Product WHERE (cname=maker) AND [(price > 100) AND (city = "Urbana")
a. Draw a logical query plan tree for the above SQL query. This plan tree should join Company and Product, then perform a selection on price and city, then project out cname and president
A: π_cname,president(σ_price>100 ∧ city='Urbana'(Company ⋈_cname=maker Product))
b. **Cost:** For the logical query plan that you draw in Part a, consider the following physical query plan. First, perform a nested loop join of Company and Product, with Company being the outer relation. Next, pipeline the result of the join into the selection operation. Finally, pipeline the result of the selection operation into the projection operation. Compute the total cost of this physical query plan
A: Since we pipeline the selection and the projection, the cost of this plan is just the cost of the join, and this is nested loop join, so we can use the formula: $|10000| + ceil(\frac{|10000|}{101-2})x|10000| = 10000 + 102 * 10000 = 1030000$
c. **Index based:** Assume that there is an index on attribute cname of relation Company, and that cname is a key for that relation. Can we replace the nested loop join operation in the physical query plan mentioned in Part b with an index based join operation? If so, which relation should be the outer relation in this index based join operation, and what should be the cost of this new physical query plan (which uses the index based join operation)
A: Yes, we can replace. It will be an index nested loop join. Product will be the outer relation. We must read in the entire Product relation, so the cost of that is 10000 pages (aka blocks). For each page, we have 5 records. For each record, we can probe the index on cname of Company, to find matching tuples in Company, and we will find just ONE matching tuple, because cname is a KEY for Company. Assuming that each tuple we find in Company resides on a separate page (a worst-case scenario), we need to read in that page. Since we have 10000*5 = 50K records in Product, and for each record we may need to read in a page in Company, the total number of pages we read in is 50K in the worst case scenario. So the total cost is 10K (to read in Products) plus 50K (of reading in Company pages). Assumption: here we also assume the index fits in memory, so there is no cost of looking up entries in the index. Note: this problem is underspecified. So as long as you make clear which assumptions you are making, and they are reasonable, then you should be fine.
d. **Push Selection:** Consider the logical query plan that you have drawn in Part a. Draw a new logical query plan that pushes selections and projections down as far as possible.
A: π_cname,president(σ_city='Urbana'(Company) ⋈_cname=maker π_maker,price(σ_price>100(Product)))
e. **Downside to Push Selection:** Outline at least one scenario where pushing selections down is not desirable. Explain why
A: If you are doing an index nested loop join between A and B, where A is the outer table, and you have an index over the inner table B, then pushing a selection down below the join, on B is not desirable, because the join then can't use the index on B.

Crash Recovery
Database has 4 elements: A,B,C,D and it just crashed. We are maintaining an undo log. Assume further that after the crash the four elements have value: A=1, B=2, C=3, and D=4. Recover the database. Clearly indicate which portion of the log you would need to inspect, and which transactions have to be executed again. Show the values of A, B, C, and D after the recovery.

A: We start from the end of the log and scan up. We see an <end ckpt> and then a bit later a <start ckpt(T1,T2)> record. So we have to process from the end of log to this <start ckpt(T1,T2)> record. Specifically, within this range, if a transaction has not been committed, then we have to undo its actions. These transactions are T6, T4, and T3. By undoing their actions, we got A=1, B=12, C=5, and D=8.

Undo log
```
<start T1>
<T1,A,3>
<start T2>
<T2,B,1>
<start ckpt(T1,T2)>  ← We know when we see this we stop, since we
finished the checkpoint.
<T1,A,4>
<start T3>
<commit T1>
<T2,B,2>
<T3,C,5>
<commit T2>
```

Undolog continued
```
<end ckpt> ← We know we now only look up to the start ckpt
<start T4>
<T4,D,8>
<start ckpt(T3,T4)>
<start T5>
<T4,D,9>
<T5,A,10>
<commit T5>
<start T6>
<T6,B,12>
```

Crash Recovery Redo Log:
Post crash state: X=1, Y=2, Z=3, U=4, V=5.
Recover from the following redo-log.

A: X=4, Y=2, Z=3, U=9, V=5.
We only want to consider transactions that have been completed, and execute the T=v (since it's new value contained in redo log) only for **completed transactions.**
My steps: cross out any TXNs not committed then just start working top down to see what gets modified.

```
<start T1>
<T1,X,3>
<start T2>
<T2,Y,1>
<start T3>
<T3,Z,5>
<T2,Y,2>
<commit T2>
<start ckpt(T1,T3)>
<T1,X,4>
<start T4>
```

```
<end ckpt>
<commit T1>
<T3,Y,2>
<T4,U,8>
<start ckpt(T3,T4)>
<start T5>
<T4,U,9>
<T5,X,10>
<commit T4>
<start T6>
<T6,Y,12>
```