Lecture 4: Query Processing

Recap

Access Methods

- Access methods are the alternative ways for retrieving specific tuples
- We covered two access methods: sequential scan and index scan
- Sequential scan is done over an unordered table heap
- Index scan is done over an ordered B-Tree or an unordered hash table
- Hash tables are fast data structures that support O(1) look-ups

Hash Tables vs. B+Trees

- Hash tables are usually **not** what you want to use for a indexing tables
  - Lack of ordering in widely-used hashing schemes
  - Lack of locality of reference ⟶ more disk seeks
  - Persistent data structures are much more complex (logging and recovery)
  - Reference
- The venerable B+Tree is always a good choice for your DBMS.
- Making a data structure thread-safe is notoriously difficult in practice.
- We focused on B+Trees but the same high-level techniques are applicable to other data structures.

Access Methods

- It is important to choose the right index for the target workload
  - Hash Table
  - B+Tree

Today's Agenda

- Query Processing
- Sorting Algorithms
- Aggregation Algorithms
- Join Algorithms
- Processing Models
- CPU and I/O Parallelism

Storage Models

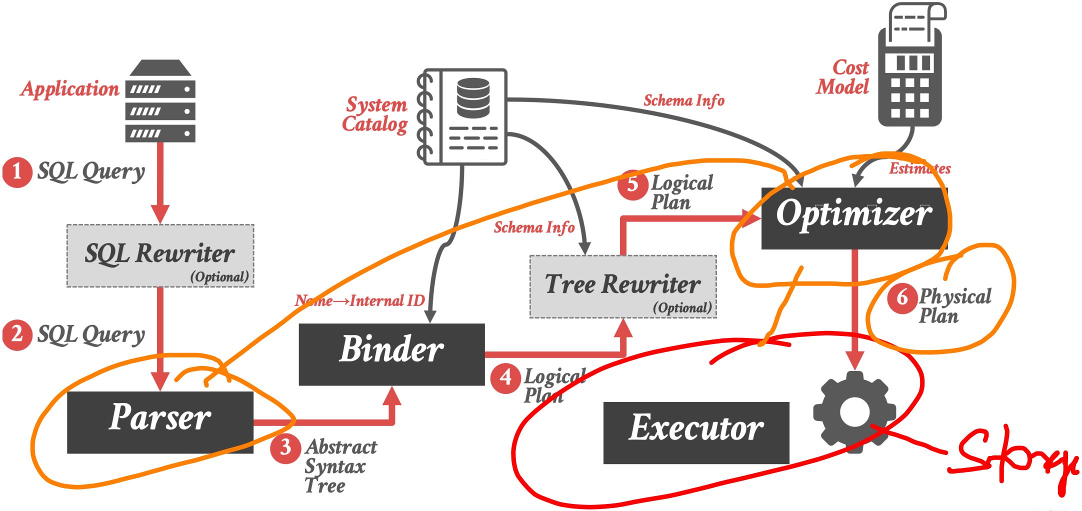
OLTP : Row : NSM

OLAP : Column : DSM

HTAP : Flexible : FSM

Query Processing

Anatomy of a Database System [Monologue]



Anatomy of a Database System [Monologue]

- Process Manager
  - Manages client connections
- Query Processor
  - Parse, plan and execute queries on top of storage manager
- Transactional Storage Manager
  - Knits together buffer management, concurrency control, logging and recovery
- Shared Utilities
  - Manage hardware resources across threads

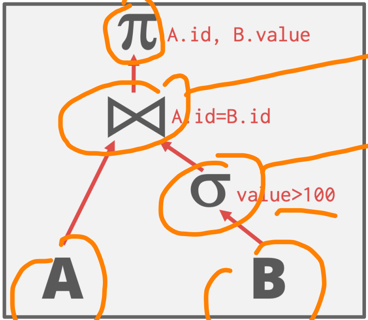
Anatomy of a Database System [Monologue]

- Process Manager
  - Connection Manager + Admission Control
- Query Processor
  - Query Parser
  - Query Optimizer (*a.k.a.*, Query Planner)
  - Query Executor
- Transactional Storage Manager
  - Lock Manager
  - Access Methods (*a.k.a.*, Indexes)
  - Buffer Pool Manager
  - Log Manager
- Shared Utilities
  - Memory, Disk, and Networking Manager

Query Plan

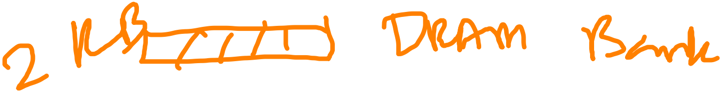
- The operators are arranged in a tree.
- Data flows from the leaves of the tree up towards the root.
- The output of the root node is the result of the query.

```sql
SELECT A.id, B.value
  FROM A, B
  WHERE A.id = B.id AND B.value > 100
```



Disk-Oriented DBMS

- We **cannot** assume that the results of a query fits in memory.
- We are going use the **buffer pool** to implement query execution algorithms that need to spill to disk.
- We are also going to prefer algorithms that maximize the amount of **sequential access**.



Sorting Algorithms

Why do we need to sort?

SET-ORIENTED

- Tuples in a table have no specific order.
- But queries often want to retrieve tuples in a specific order.
  - Trivial to support duplicate elimination (`DISTINCT`).
  - Bulk loading sorted tuples into a B+Tree index is faster.
  - Aggregation (`GROUP BY`).

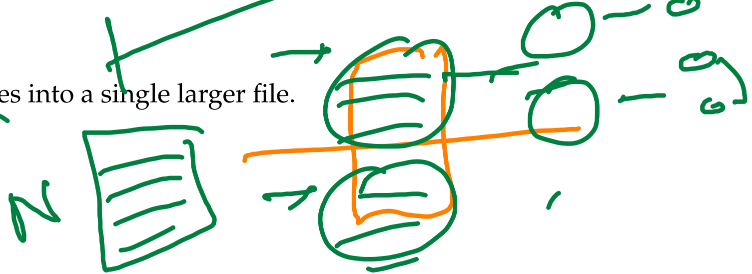
ORDER BY

Sorting Algorithms

- If data fits in memory, then we can use a standard in-memory sorting algorithm like **quick-sort**.
- If data does not fit in memory, then we need to use a technique that is aware of the cost of writing data out to disk.

External Merge Sort

- Divide-and-conquer sorting algorithm that splits the data set into separate **runs** and then sorts them individually.
- **Phase 1 – Sorting**
  - Sort blocks of data that fit in main-memory and then write back the sorted blocks to a file on disk.
- **Phase 2 – Merging**
  - Combine sorted sub-files into a single larger file.

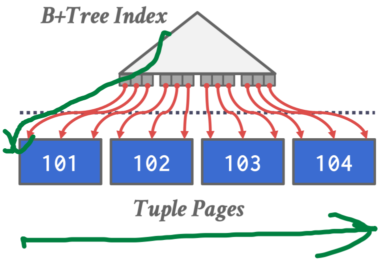


Using B+Trees for Sorting

- If the table that must be sorted already has a B+Tree index on the sort attribute(s), then we can use that to accelerate sorting.
- Retrieve tuples in desired **sort order** by simply traversing the **leaf pages** of the tree.
- Cases to consider:
  - Clustered B+Tree
  - Unclustered B+Tree

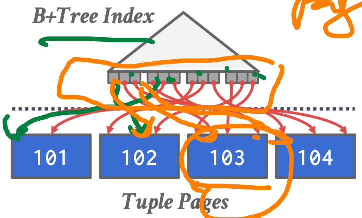
Case 1 – Clustered B+Tree

- Traverse to the left-most leaf page, and then retrieve tuples from all leaf pages.
- This is always better than external sorting because there is no computational cost and all disk access is sequential.



Case 2 – Unclustered B+Tree

- Chase each pointer to the page that contains the data.
- This is almost always a bad idea. In general, one I/O per data record.



Aggregation Algorithms

Aggregation

SUM(SALARY) GROUP BY

DEPT_ID

- Collapse multiple tuples into a single scalar value.
- Two implementation choices:
  - Sorting
  - Hashing

Sorting Aggregation

```sql
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
 ORDER BY cid
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove Columns

cid
15-445
15-826
15-721
15-445

Sort

cid
15-445
15-445
15-721
15-826

Sorting Aggregation

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
 ORDER BY cid
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove Columns

cid
15-445
15-826
15-721
15-445

Sort

cid
15-445
~~15-445~~
15-721
15-826

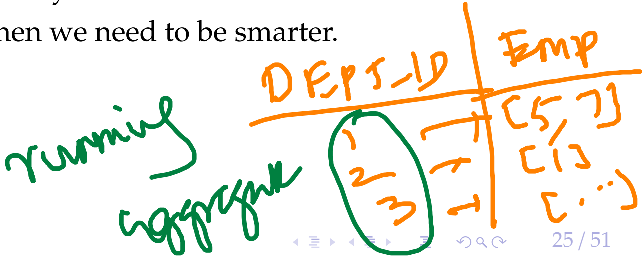
Eliminate Dupes

Alternatives to Sorting

- What if we **do not** need the data to be ordered?
  - Forming groups in `GROUP BY` (no ordering)
  - Removing duplicates in `DISTINCT` (no ordering)
- Hashing is a better alternative in this scenario.
  - Only need to remove duplicates, no need for ordering.
  - May be computationally cheaper than sorting.

Hashing Aggregate

- Populate an **ephemeral hash table** as the DBMS scans the table.
- For each record, check whether there is already an entry in the hash table:
  - `GROUP BY`: Perform aggregate computation.
  - `DISTINCT`: Discard duplicates.
- If everything fits in memory, then it is easy.
- If the DBMS must spill data to disk, then we need to be smarter.



Join Algorithms

Why do we need to join?

1NF

2NF . . .

- We **normalize** tables in a relational database to avoid unnecessary repetition of information.
- We use the join operator to reconstruct the original tuples without any information loss.

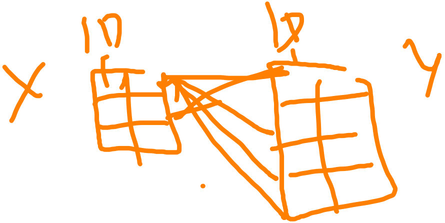
Join Algorithms

- We will focus on combining **two tables** at a time with **inner equi-join** algorithms.
  - These algorithms can be tweaked to support other types of joins.
- In general, we want the smaller table to always be the left table (**outer table**) in the query plan.

Join vs Cross-Product

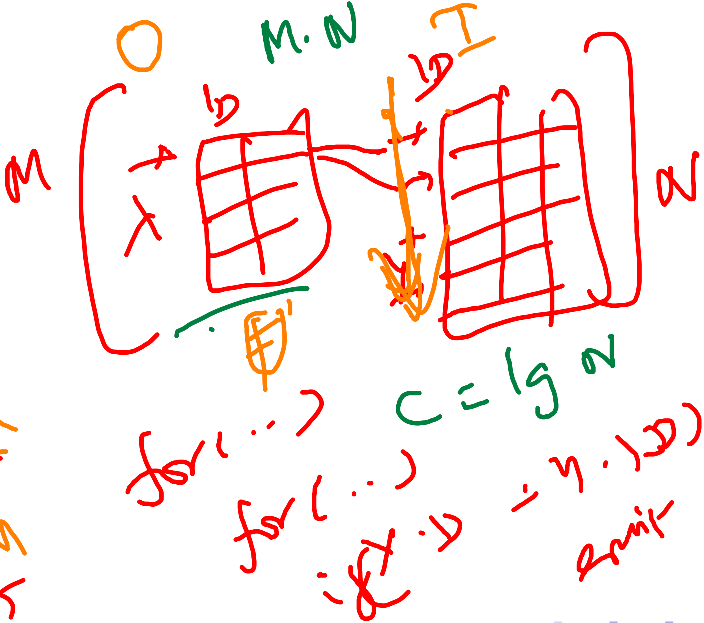
Django ORM

- $R \bowtie S$ is the most common operation and thus must be carefully optimized.
- $R \times S$ followed by a selection is inefficient because the cross-product is large.
- There are many algorithms for reducing join cost, but no algorithm works well in all scenarios.



Join Algorithms

- Nested Loop Join
  - Naïve
  - Block
  - Index
- Sort-Merge Join
- Hash Join



Join Algorithms: Summary

Join Algorithm	IO Cost	Example
Simple Nested Loop Join	M + (m x N)	1.3 hours
Block Nested Loop Join	M + (M x N)	50 seconds
Index Nested Loop Join	M + (M x C)	Variable
Sort-Merge Join	M + N + (sort cost)	0.75 seconds
Hash Join	3 x (M + N)	0.45 seconds

Processing Models

Processing Model

- A DBMS's **processing model** defines how the system executes a query plan.
  - Different trade-offs for different workloads.
- **Approach 1:** Iterator Model
- **Approach 2:** Materialization Model
- **Approach 3:** Vectorized / Batch Model

Iterator Model

- Each query plan operator implements a `Next` function.
  - On each invocation, the operator returns either a single tuple or a null marker if there are no more tuples.
  - The operator implements a loop that calls next on its children to retrieve their tuples and then process them.
- Also called **volcano** or **pipeline** model.

Iterator Model

- This is used in almost every DBMS. Allows for tuple **pipelining**.
- Some operators have to block until their children emit all of their tuples.
- These operators are known as **pipeline breakers**
  - Joins, Subqueries, Order By
- Output control (*e.g.*, `LIMIT`) works easily with this approach.
- **Examples:** SQLite, MySQL, PostgreSQL

Materialization Model

Memory

- Each operator processes its input **all at once** and then emits its output all at once.
  - The operator "materializes" its output as a single result.
  - The DBMS can push down **hints** into to avoid scanning too many tuples (*e.g.*, `LIMIT`).
  - Can send either a materialized row or a single column.
- The output can be either whole tuples (NSM) or subsets of columns (DSM)

Materialization Model

- Better for OLTP workloads because queries only access a small number of tuples at a time.
  - Lower execution / coordination overhead.
  - Fewer function calls.
- Not good for OLAP queries with large intermediate results.
- **Examples:** MonetDB, VoltDB

DSM

In-memory

Vectorization Model

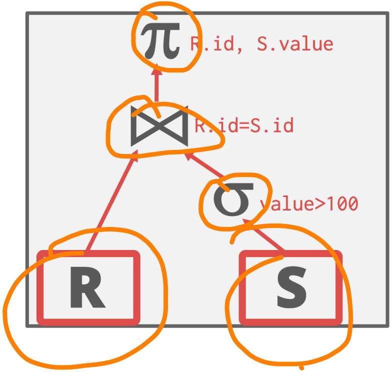
- Like the Iterator Model where each operator implements a Next function in this model.
- Each operator emits a **batch of tuples** instead of a single tuple.
  - The operator's internal loop processes multiple tuples at a time.
  - The size of the batch can vary based on hardware or query properties.
  - Useful in in-memory DBMSs (due to fewer function calls)
  - Useful in disk-centric DBMSs (due to fewer IO operations)

Vectorization Model

- Ideal for OLAP queries because it greatly reduces the number of invocations per operator.
- Allows for operators to use vectorized (SIMD) instructions to process batches of tuples.
- **Examples:** Vectorwise, Snowflake, SQL Server, Oracle, Amazon RedShift

Access Methods

- An **access method** is a way that the DBMS can access the data stored in a table.
  - Located at the bottom of the query plan
  - Not defined in relational algebra.
- Three basic approaches:
  - Sequential Scan
  - Index Scan
  - Multi-Index / "Bitmap" Scan

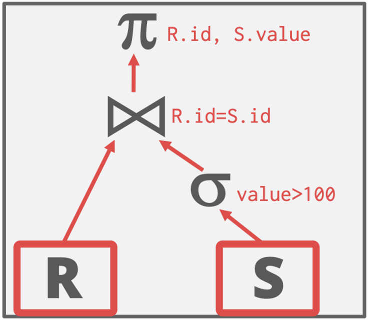


CPU and I/O Parallelism

Query Execution

- We discussed about how to compose operators together to execute a query plan.
- We assumed that the queries execute with a single worker (*e.g.*, thread).
- We now need to talk about how to execute with multiple workers.

```
SELECT R.id, S.cdate
  FROM R, S
  WHERE R.id = S.id AND S.value > 100
```



Why care about Parallel Execution?

- Increased performance.
  - Throughput
  - Latency
- Increased responsiveness and availability.
- Potentially lower **total cost of ownership** (TCO).

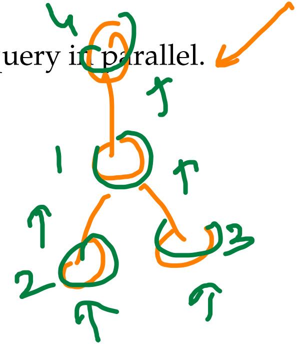
Scale-up
vs
Scale-out

Parallel Execution

- CPU Parallelism
- I/O Parallelism

Inter- VS. Intra-Query Parallelism

- **Inter-Query:** Different queries are executed concurrently.
  - Increases throughput & reduces latency.
- **Intra-Query:** Execute the operations of a single query in parallel.
  - Decreases latency for long-running queries.



Observation

Disk

- Using additional processes/threads to execute queries in parallel won't help if the disk is always the main bottleneck.
  - Can make things worse if each worker is reading different segments of disk.

locality

I/O Parallelism

- Split the DBMS installation across multiple storage devices
  - Multiple Disks per Database
  - One Database per Disk
  - One Relation per Disk
  - Split Relation across Multiple Disks

Conclusion

Parting Thoughts

- Access methods are the alternative ways for retrieving specific tuples
- Hashing is almost always better than sorting for operator execution.
- Caveats:
  - Sorting is better on non-uniform data.
  - Sorting is better when result needs to be sorted.
- Good DBMSs use either or both.

Parting Thoughts

- The same query plan be executed in multiple ways.
- A DBMS's **processing model** defines how the system executes a query plan.
- (Most) DBMSs will want to use an index scan as much as possible.
- Parallel execution is important.
- (Almost) every DBMS supports this.
- This is really hard to get right.
  - Coordination Overhead
  - Scheduling
  - Concurrency Issues
  - Resource Contention

Next Class

- Logging and Recovery Protocols