CS 143

Warren Kim

Contents

Lecture 10

1.1 OnLine Transaction Processing

Definition: OnLine Transaction Processing (OLTP)

OnLine Transaction Processing is a type of data system and is designed for frequent interactive use and optimizes *random* access with low latency and can be used in production. Each read/write is a result of a *transaction* carries out by the user. RDBMS are a *type* of OLTP.

OLTP's typically involve simple queries (e.g. `SELECT`, basic `JOIN`'s). They

→ optimize for quick, random access.

→ are based on *transactions*, which are individual events that are stored as *rows* in an RDBMS.

→ strive to minimize redundancy and require joins to exploit relationships in data; that is, they are normalized.

→ are designed for production use: data is typically accessed by online applications or users.

→ abstract data as a 2-dimensional representation called a *table*.

1.2 OnLine Analytical Processing and Data Warehousing

Definition: OnLine Analytical Processing/Data Warehousing (OLAP/DW)

OnLine Analytical Processing is a system that is not meant to be accessed in production, and only by internal users. It is *read-only* and allows for *very fast, low latency* reads of data.

Data Warehouses are an implementation of OLAP concepts.

Typically, only other automated systems (ETL) write into OLAP's. They are usually used for analytics by internal users (e.g. data scientists, business analysts, etc.). OLAP/DW's are **completely** different systems from RDBMS/OLTP, ***but*** both typically use SQL.

OLAP's use an **ad hoc** write system which writes in batches, since incremental writes are slow. Pure OLAP systems aren't very common anymore, but many of the concepts are still used today in relational databases or data warehouses.

The user still "sees" and "works" with tables. However in OLAP, data is conceptualized as a cube with rows, columns, and depth. Each cell/subcube is referred to as a **measure**.

Example

Suppose we have a cube where the length represents shipping date (e.g. 2018), height represents the product (e.g. tablet), and the depth represents a location (e.g. US West Coast). Then the measure of this cube is revenue, and looks like

Location	Product	Shipping	Revenue
US West Coast	Tablet	2018	$5M

OLAP is typically used for *batch processing* and not for production systems. They are optimized for low latency reads (`SELECT`) of **aggregated** or **precomputed** data. We typically perform a precomputed `JOIN` during a low usage period (e.g. overnight) via an automated system like an ETL job and then read from the OLAP throughout the day.

1.2.1 ETL Jobs in OLAP's:

An ETL job transfers either all of the data (inefficient) or new data (since the last update). But this is very costly to the production database (if it's being used). Additionally, we may lose some data if inserts/updates in the OLTP are processed during the transfer. Since we typically work with denormalized tables and aggregates in OLAP's, this is usually okay.

Addressing Performance: In order to address the performance issue with ETL jobs executing against the production database, we can have a replica database and run the ETL job against the replica. We can keep the replica in sync with the production database by performing **dual writes**, where, on each write to the production database, we also write to the replica.

1.2.2 Exploded Tables

The first thing OLAP does when the data is loaded is to perform various aggregates and construct different dimensions (groups). These functions can be:

→ specified *a priori* by a data engineer.

→ automatically inferred by the system. If we have n grouping columns, then we have $\sum_{k=0}^{n}\binom{n}{k} = 2^n$ dimensions (groups).

In either case, this process is time consuming and is CPU intensive, hence why it's usually computed overnight. You can have multiple data warehouses for the same database. It can be split by department, region, etc.

Data warehouses also typically store **exploded tables**, or denormalized tables with precomputed `JOIN`'s for low latency reads. There is a catch; we will be working with *outdated* data most of the time since we only update the data warehouse periodically and not in real time. Typically, these tables are *append-only*; i.e. no moditications or deletions.

Separation of Concerns: We have a separation between RDBMS and data warehouses because we cannot always trust the user.

1.2.3 OLAP Operations

There are five operations in OLAP, now implemented in RDBMS. Throughout each operation, assume we are working with a data warehouse that stores product information where the length is the locations (e.g. NA), height is the product (e.g. wireless mouse), and depth is the time (e.g. 2003).

Slice: A **slice** selects *one* predominant dimension from a cube and returns a new sub-cube. For example, if we want product = wireless mouse, we get a rectangular representation of data containing only wireless mice, but across all locations and time. In SQL, this is similar to `WHERE`.

Dice: A **dice** selects multiple values from multiple dimensions and returns a new sub-cube. For example, location = NA *and* product = Nokia. In SQL, this is similar to a `WHERE` with `AND`'s.

Aside

The distinction between slice and dice is typically only in theory. We typically only use slice.

Rollup: A **rollup** computes aggregates across all levels of a hierarchical attribute. For example, aggregating the sales for a particular day $\subseteq$ month $\subseteq$ quarter $\subseteq$ year $\subseteq \cdots$. In SQL, it looks like

```
SELECT year, month, day, SUM(sales) AS total_sales
FROM hourly_sales
GROUP BY ROLLUP(year, month, day);
```

where, from left to right, we have least granular → most granular; i.e. year $\supseteq$ month $\supseteq$ day. Generally, `ROLLUP(A, B, C)` where

$$\texttt{A} \supseteq \texttt{B} \supseteq \texttt{C}$$

Example

The output of the following query:

```
SELECT year, month, day, SUM(sales) AS total_sales
FROM hourly_sales
GROUP BY ROLLUP(year, month, day);
```

looks like

year	month	day	total_sales
2020	April	21	200
⋮	⋮	⋮	⋮
2020	December	31	842
2020	April	NULL	2662
⋮	⋮	⋮	⋮
2020	December	NULL	8412
2020	NULL	NULL	126830
NULL	NULL	NULL	2526124

Note: A **cube** in SQL produces all subsets of group columns.

Drill Down: A **drill down** extracts aggregates at a finer level of granularity. For example, going from an annual aggregate *down* to a week or day aggregate.

Example

Dashboarding is an example of using OLAP operations. For example, we can take a Facebook sentiment analysis.

→ Take a **slice** where company = Walmart.

→ Filter by gender by **dicing** where company = Walmart *and* gender = woman.

→ Choosing a state uses a **drill down** since states $\supseteq$ counties $\supseteq$ cities $\supseteq \cdots$.

→ If we want to zoom out one level, we use **rollup**.

Pivot: A **pivot** converts data from *long* format to *wide* format and vice-versa.

Example

Long Format: In a **long** format, the table is more flexible so it's easier to add another row. There are also no `NULL`'s. However, we now have redundancy, which implies that this table is denormalized. It may also be harder to understand.

uid	full_name	assignment	mark
012345678	Schmoe, Joe	hw1	100
012345678	Schmoe, Joe	hw2	99
012345678	Schmoe, Joe	hw3	89
876543210	Bruin, Joe	hw1	14
876543210	Bruin, Joe	hw2	79
876543210	Bruin, Joe	hw3	87
876543210	Bruin, Joe	hw4	79
424242424	Block, Gene	hw1	81
424242424	Block, Gene	hw2	37
424242424	Block, Gene	hw3	89

$$\Longleftrightarrow$$

Wide Format: In a **wide** format, there is less redundancy so it's easier to understand, but now we have the possibility of `NULL`'s and the format is inflexible (since we would have to use an `ALTER TABLE`).

uid	full_name	hw1	hw2	hw3	hw4
012345678	Schmoe, Joe	100	99	89	
876543210	Bruin, Joe	14	79	87	79
424242424	Block, Gene	81	37	89	

Note: Long format is more common in analytics and in general.

Long to Wide: We can use a searched case statement to collapse the table from a long format into a wide format:

```
SELECT uid, full_name,
       SUM(CASE assignment WHEN ’hw1’ THEN ’mark’ ELSE 0 END) AS hw1,
       SUM(CASE assignment WHEN ’hw2’ THEN ’mark’ ELSE 0 END) AS hw2,
       SUM(CASE assignment WHEN ’hw3’ THEN ’mark’ ELSE 0 END) AS hw3,
       SUM(CASE assignment WHEN ’hw4’ THEN ’mark’ ELSE 0 END) AS hw4
FROM homework_grades
GROUP BY uid, full_name;
```

Wide to Long We can use a union to collapse the table from a long format into a wide format:

```
SELECT uid, full_name, ’hw1’, hw1 FROM wide_format
UNION
SELECT uid, full_name, ’hw2’, hw2 FROM wide_format
UNION
SELECT uid, full_name, ’hw3’, hw3 FROM wide_format
UNION
SELECT uid, full_name, ’hw4’, hw4 FROM wide_format;
```

1.2.4 Schemas in Data Warehouses

Data warehouses can be used for joins, but the types of tables in a DW restrict this to:

→ A **fact** table which contains quantitative data to be analyzed. They are typically denormalized.

→ A **dimension** table which contains data about attributes of each of these facts.

There are three common schemas in data warehouses.

Star Schema: The **star schema** contains a single fact table along with several dimension tables that must be joined to it to get a result.

Snowflake Schema: The **snowflake schema** contains a single fact table along with several dimension tables that must be joined to it to get a result. Unlike the star schema, the dimension table references other dimension table(s) to be able to fully describe the fact; i.e. multiplie dimension tables need to be joined to describe the fact.

Galaxy Schema: The **galaxy schema** contain multiple fact tables that *share* dimension tables among them. It is the closest to RDBMS. The multiple fact tables are connected via the dimension table(s) they share.

1.2.5 DuckDB

DuckDB is an OLAP data store. It can be used standalone. It is in memory and inprocess, optimized for analytics, uses teh **columnar** data model, and doesn't require a server.

1.2.6 Summary

OLAP is optimized for fast access to aggregated data or denormalized tables. The data is multidimensional and dimensions, interactions, and aggregations on them are pre-computed. Data is loaded in bulk (typically overnight), which may lead to it being outdate but that's usually not a concern. There are no online modifications and have very fast read times. Data can be changed from long to wide format (and vice-versa) via pivoting. Most importantly, many concepts from OLAP apply to RDBMS.

1.3 Data System Architecture

Depending on the type of work that needs to be done with the data, we may opt for an OLTP or an OLAP. Software engineers and end-users need fast read/write access to interact with the application at hand; i.e. production data should be stored in an OLTP. Analysts, managers, and scientists typically don't need access to real-time data. Therefore, they should use and OLAP.

1.3.1 Replication

One common architecture is to have multiple copies of the same database:

→ A **production database** that gets its data from another system like an ETL job,

→ a **development database** that is used to test new features/process diagnotics,

→ a **read-only database**,

→ optionally, a **R&D database**.

The databases are kept in sync either via an ETL job, a message bus that multiplexes data operations, or advanced replication options.

1.3.2 Multiple Systems

We may have multiple systems that contain the same/similar data:

→ DBMS for production data,

→ a key-value store or cache for frequently used data,

→ data warehouses for dashboarding, reporting, and fast access to aggregates,

→ a search engine for quickly locating records,

→ a big data system for ETL jobs.

1.4 Data Lakes

Definition: Data Lake

A **data lake** is a single system that contains raw, unprocessed data and has no schema. They can be thought of as a big repository of raw data.

Clean data lakes require a lot of upkeep, curation, and management. Data lakes may bloat with unnecessary data since developers will dump any and all data into them, thinking they will need it one day. Thus, data lakes may sometimes be called “data swamps” or “data graveyards”.

We want to be able to efficiently access and process data from a data lake, allow data sharing between systems and users (authorization model), efficiently catalog/index the contents of a data lake, and be (program) language agnostic.

1.5 Comparisons

	RDBMS/Database	Data Warehouse	Data Lake
Data	Normalized, clean tables	Redundant, aggregated or denormalized	Any type/Raw data
Schema	Fixed and Strict	Fixed and checked on bulk entry	None
Price	Good random access	Fast reads	Bad Performance
Performance	Free → Expensive	Expensive	Cheap/Free
Data Quality	Good	Curated, May be outdated	Bad
Users	App/SWE	Analysts	Who knows
Analytics	Not Good	Batch processing, Good	MapReduce/Big Data

Lecture 11

2.1 Distributed Systems

There are two main ways to design a distributed data system.

Replication: In **replicated** systems, all nodes have the same data. This is good for load balancing, the system is fault tolerant $\implies$ highly available, but *mostly* consistent.

Example

In a *centralized* server with three nodes A, B, C, when A executes a write, it also writes to the master data store, which will write to B, C. If B tries to read data that hasn't been replicated yet, it will ask the master data store which will then return the data.

In a *decentralized* server with three nodes A, B, C, when A executes a write, it also broadcasts to B, C. This is also known as a **gossip model**.

Sharding: In **shareded** systems, different nodes contain different data. Whether or not each node is disjoint from another is up to design. This is good for low latency (within your local node).

Example

In a *centralized* server with three nodes A, B, C, when A executes a write, nothing else happens. If B tries to read data that is written in A, B asks the master data store which will then ask A for the data.

In a *decentralized* server with three nodes A, B, C, when A executes a write, nothing else happens. If B tries to read data that is written in A, B asks A for the data.

2.1.1 Distributed Filesystem Architecture

A distributed filesystem is decomposed into m machines, where each machine m_i may have multiple hard disks. In either case, we allocate space on the disk for a directory that is part of the distributed file system across the m machines.

2.2 Big Data

Definition: Big Data

Big data is data that is too large to fit into RAM. It can be on the order of terabytes or petabytes.

The purpose of a big data system is to maximize the usage of compute resources to increase the amount of work we can do per unit time via *parallelism*. Note that this work involves processing large amounts of data that may or may not be structured.

Aside

One common misconception is that parallelism makes data processing faster. This need not be true. Parallelism increases the maximum *throughput*, but if there is too much overhead, parallelism can be slower.

2.3 MapReduce (in Hadoop)

Throughout this section, we will use a word counting example.

2.3.1 MapReduce Operations

Input: The `InputFormat` will define how records are laid out in files and how to divide groups of records into **splits**, which define the level of parallelism.

Map: Once we partition the data into splits, each split will perform the **map** task on a record. In this example, we can either

(1) output the key-value pair $(w, 1)$ for every single occurrence of every word w. This uses a lot of disk I/O but little RAM.

(2) output a key-value pair (w, c) where c is the count for one word w. This uses little disk I/O but more RAM.

We can do (2) in the map phase or we can implement it in the **combiner** that performs an aggregation right after the map phase. The map tasks run on each record in parallel and are independent of one another.

Partition: After each map task completes, we **group** the key-value pairs by **key**; all key-value pairs with the same key will be sent to the same reducer task in the **reduce** phase. The partitions are **local** to each map task. Each key-value pair is assigned to a partition using a *hash function* applied ot the key followed by a *modulo* n where n is the number of reduce tasks. We can implement our own hash function if need be.

Note: Up to the partition phase is **map-side**.

Shuffle: The partitions from each map task are **shuffled** across the network to a series of reducers, which is chosen by the resource manager or job tracker.

Sort: Each partition is sent to the reducer which **sorts** the key-value pairs by key. The values are then grouped by key: `(key, [list of values])`. For example, `(Car 2), (Car 1)` → `(Car, [2, 1])`.

Reduce: Each reducer executes a **reduce** function on the list of values for each key. In this example, we can apply a summation, or `(Car, SUM([2, 1]))` → `(Car, 3)`.

Note: We execute all of the reduce functions in parallel. Since the partitioner sent all of the keys to the correct reducer, we should not have duplicate keys sent to different reducers. The one exception is if there is an imbalance in the reducers; e.g. the word "the" is more common than the word "xylophone", so we might send some "the"s to the x-reducer via a custom hash. This requires a second reduce phase to aggregate all of the "the"s.

Output: Each reduce task outputs a single file. The total number of output files is equal to the number of reducers. We can use `OutputFormat` store the output in an alternate format.

Note: From the sort phase to the output phase are **reduce-side**.

2.3.2 Infrastructure and Example

MapReduce functions on each input split *independently*. It is a **shared-nothing** model and is **embarrassingly parallel**. The following Python script is an example of a MapReduce job:

```
#!/user/bin/env python
'''mapper.py'''

import sys

for line in sys.stdin:          # input comes from stdin.
    line = line.strip()         # remove leading/trailing whitespace.
    words = line.split()        # split the line into words.

    for word in words:          # iterate across each word.
        print(f'{word}\t{1}')   # (w, 1)
```

```
#!/user/bin/env python
'''reducer.py'''

import sys
from operator import itemgetter

for line in sys.stdin:   # input comes from stdin.
    line = line.strip()  # remove leading/trailing whitespace.
    word, count = line.split('\t', 1)  # split the line into (w, c)
    count = int(count)

    if curr_word == word:
        curr_count += count
    else:
        if curr_word:
            print(f'{curr_word}\t{curr_count}')  # (w, c)

        curr_count = count
        curr_word = word

if curr_word == word:  # last word
    print(f'{curr_word}\t{curr_count}')  # (w, c)
```

To run, do `cat fox.dat | ./mapper.py | sort | ./reducer.py`.

2.4 SQL in MapReduce

The **map** phase is a σ/`WHERE` The **reduce** phase is a γ. The partition/sort is a `GROUP BY`, and the reducer is an aggregation function (e.g. `SUM`).

2.4.1 Subqueries

Subqueries can encourage parallelism for MapReduce jobs in SQL. The optimizer will order the computeations in a way that the result of the subquery materializes before the outer query.

2.4.2 Efficiency

A join in relational algebra is one operation, but in MapReduce, it's two operations. Representing relational algebra as a series of map and reduce steps is inefficient.

2.5 Apache Spark

Rather than dealing with files and filesystems, we deal with datasets called Resilient Distributed Datasets (RDD's) that

→ can be any format.

→ are resilient; if one node fails, there is a replica somewhere else.

→ are distributed; parts of the data can be on differen tnoes in the system.

→ are immutable.

2.5.1 Spark vs. Hadoop

Spark is usually better than Hadoop since:

→ computations are stored in RAM until we explicitly request to persist them to disk (`collect()`).

→ Spark uses lazy evaluation and builds a DAG. This makes it easier to handle iterative problems.

→ Spark has been measured to be up to 100x faster than Hadoop when using RAM, and 10x faster when using disk/HDFS.

However, disk is cheap (but slow), so use Hadoop. If you can afford RAM, use Spark.

2.5.2 Transformations

Transformations return immediately but are not persisted to the disk. These commands build the DAG. Some examples include:

→ `sample()`

→ `reduceByKey()`

→ `join() filter()`

→ `sort()`, `sortByKey()`

→ `groupBy()`, `distinct()`

→ `union()`, `intersection()`

→ `map()`, `flatMap()`, `mapPartitions()`

2.5.3 Actions

The pipeline only executes when an **action** is executed. These commands execute the DAG. Some examples include:

→ `count()`, `countByKey()`

→ `head()`, `first()`, `take(n)`

→ `reduce(func) foreach(func)`

→ In DataFrame/DataSet, `show()`

→ `collect()`, `collectAsList()`, `collectAsMap()`

→ `saveAsTextFile()`, `saveAsSequenceFile()`, `saveAsObjectFile()`

2.5.4 Examples

Word Counts in Python and Scala

```
text_file = sc.textFile('hdfs://...')
counts = (text_file.flatMap(lambda line : line.split(' '))
                   .map(lambda word: (word, 1))
                   .reduceByKey(lambda a, b: a + b))
counts.saveAsTextFile('hdfs://...')
```

```
val textFile = sc.textFile('hdfs://...')
val counts = textFile.flatMap(line => line.split(' '))
                     .map(wrod => (word, 1))
                     .reduceByKey(_ + _)
counts.saveAsTextFile('hdfs://...')
```

Computing π in Python and Scala

```
import random
def inside():
    x, y = random.random(), random.random()
    return x * x + y * y < 1
count = (sc.parallelize(range(NUM_SAMPLES))
           .filter(inside).count())
print(f'pi is roughly {4.0 * count / NUM_SAMPLES')
```

```
val count = sc.parallelize(1 to NUM_SAMPLES)
              .filter { _ =>
                        val x = math.random
                        val y = math.random
                        x * x + y * y < 1
                      }.count()
println(s'pi is roughly ${4.0 * count / NUM_SAMPLES}')
```

2.5.5 Spark SQL

Spark is not an RDBMS and joins in Spark SQL suck. Below is an example of the word counts example in Spark SQL.

Note: `HAVING` is only available in pure SQL.

```
data = spark.read.format('csv').option('header', 'true').load('data.csv')
data.printSchema() # prints the schema to the data frame.
word_counts = (data.withColumn('cleaned_text', cleantext(data))
                   .select('cleaned_text')
                   .withColumn('token', explode('cleaned_text'))
                   .select('token')
                   .where("token != 'the'")
                   .groupBy('token')
                   .count()
                   .orderBy(desc('count'))
                   .head(20))
word_counts.explain()
word_counts.write.csv('...')
```

Lecture 12

3.1 (Unbounded) Streaming Data Systems

Definition: Lambda Architecture

The **lambda architecture** processes massive quantities of data by using both batch and real-time/stream processing (speed).

Examples of streams include:

→ `stdin/out` and pipes

→ TCP connections

→ Realtime audio/video and sensor measurements

→ Realtime data (e.g. tweets, stock prices, etc.)

Data streams (events/messages)[1] can be of any format (e.g. plaintext, JSON, etc.). Each event is written once, but may be read multiple times, and may have a key to identify itself (e.g. timestamp, IP addr, etc.) and/or other identification.

We *can* store these in a database, but usually we want to act on events as they come, and possibly without storing them.

3.1.1 Producer/Consumer

There are two ends or "taps" for processing events:

Definition: Producer and Consumer

The **producer** (publisher, sender) creates an event.

The **consumer** (subscriber, recipient) receives and processes *or* hands off an event.

If we want to use an RDBMS, the producers would generate events (rows) continuously into the database while the consumers constantly poll the database for new events. However, polling (querying) is very expensive and increaeses latency.

So, rather than "pulling" from a data store, we want to "push" notifications onto the consumers. RDBMS can use triggers to do this, but it's limiting.

3.1.2 Messsage Bus

A **message bus** sits between producers and consumers, where each producer is connected to every consumer and vice-versa (creating a bipartite graph). Some issues of this approach is:

→ Producers must be hardcoded to the consumers.

→ The producers are all independent, so if one producer goes down, there is no way to compensate

→ If a consumer becomes unavailable, we have to manually spin up a new one.

→ Consumers could be overwhelmed; i.e. producers will produce faster than consumers can consume.

[1] I will refer to them as "events".

3.1.3 Data Integrity

We discuss two use cases:

(1) Replicate to RDBMS.

(2) Compute an aggregate metric (over unbounded data).

Data Loss: In (1), it is *bad*. In (2), if the messages are missing *at random*, it is (usually) *not bad*.

Duplicate Messages: In (1), it is *not bad* assuming the message has a key. In (2), if receiving duplicates *at random*, it is (usually) *not bad*.

Out of Order Messages: In (1), it is *not bad*. In (2), it is *not bad* unless we do sequence mining.

TCP or UDP: Depending on the use case, we may want a TCP or UDP connection. If we to keep a connection alive between a producer and a consumer along with ordered packets and guaranteed delivery (retransmission), use TCP. Otherwise, UDP is good enough.

Classifying Streaming Architectures: We can classify systems based on how they react to the following:

→ What happens when if the producers send messages faster than the consumers can process them? Do we drop or buffer messages? Do we apply backpressure?

→ What happens if one or more nodes (particularly consumers) go offline?

3.2 Message Broker

Definition: Message Broker

The **message broker** sits inside the messgae bus and has a **message queue**. The producers and consumers interact with the message broker.

When a message is sent to the broker, it enqueues the message and sends an ACK back to the producer. It then tries to send it to the consumer(s). The broker then waits for an ACK from the consumer before dequeuing the message. Otherwise, it stays in the buffer (queue).

While it's usually okay ot lose *some* messages, we want to minimize the loss. One way we cna do this is using an ACK system.

3.2.1 Unavailable Consumer

When conumers become too busy or become unavailable, the system can either apply flow control or spin up more consumers. If we are willing to tolerate *some* message loss, we ca:

→ Delete the *oldest* message(s) from the queue. This is okay if we are working with relevant "current" data.

→ Delete *random* message(s) from the queue. This is okay if we are computing aggregates.

3.2.2 Producer to Broker

Supose our TTL is 5 seconds. If the message is queued successfully on the broker B, the broker sends an ACK to the producer P and P dequeues the message. If the message is lost, P retransmits. If the message is corrupted in transit, B either does nothing or sends a NACK. P then retransmits up to TTL. If ACK gets lost, P assumes B didn't receive the message, so it will retransmit. This implies that B may receive duplicates.

3.2.3 Broker to Consumer

Suppose our TTL is 5 seconds. If the message is processed *successfully*, the consumer C sends an ACK and the broker B dequeues the message. If the message is lost, B retransmits. If the message is corrupted in transit, C either does nothing or sends a NACK. B then retransmits up to TTL. If ACK gets lost, B assumes C didn't receive the message, so it will retransmit. This implies that C may receive duplicates and B may back up with messages.

3.2.4 Unreliable Broker

If we have an unreliable broker B, the producer P should dequeue the message when it receives an ACK from the consumer C via B. Some sources of message loss include: lossy networks, P, B, C is offline or overflows with messages.

3.2.5 Consumers

Multiple consumers can read the same topic stream. But the broker determines which one gets the message in two ways.

Load Balancing: The broker will choose one consumer to receive the message either arbitrarily or based on some shard/partition key.

Fan-out: The broker will deliver the message to *all* consumers in particular groups. Then each node may do something different with the message. This is similar to multiplexing (e.g. dual writes).

Purpose: No single data system can meet all use cases. Therefore, we may want to have the following architecture:

→ OLTP and RDBMS for production uses.

→ OLAP/DW for analytics.

→ A search engine for quick lookups.

→ A cache for frequently accessed elements.

All of these systems need to stay in sync, and a streaming system is one way to accomplish this. Another is by using an ETL job.

3.3 Stream Operations

We can do several things with streams, including:

→ writing events to an RDBMS or other data systems to keep them in sync.

→ pushing events to users (notifications) or dashboarding.

→ processing the stream and trurn it into other data/another stream.

→ computing aggregate summaries of the stream.

3.4 Aggregates

We do not have access to all of the data in a stream, so we typically process data either over a particular window w of time or over time. The computations can be either exact or approximations.

3.4.1 Windows

The consumers maintain a memory buffer containing a fixed number of messages. We then compute an aggregation over the parts of the buffer or at specific time intervals over the entire buffer. There are various types of computation windows.

Tumbling Window: Tumbling windows are non-overlapping windows of time and are consecutive. This means the intervals are disjoint. For example, if we compute at the start of every hour over a time length of one hour, our intervals are disjoint.

Hopping Window: Hopping windows are fixed-length periods, but time intervals may overlap. This way, there are no hard breaks in the windows since the intervals are not disjoint, giving us *some* continuity. For example, if we compute at the start of every hour over a time length of one hour and fifteen minutes, we have a fifteen minute overlap.

Sliding Window: Sliding windows are constructed with some time period ε around each event.

Session Window: Session windows define an entire session from the first event of a particular type to the last event (inefficient).

Note: Sliding and session window sizes are defined by the events.

3.4.2 Stragglers

Some straggler events that should have been processed in w_{t-1} may not arrive until some time after w_{t-1} has been closed and processed. If we want to compute some metric over the window, we can:

(1) ignore stragglers.

(2) keep the w_{t-1} open for some δ time to capture stragglers.

(3) offer a correction if we store the metric associated with w_{t-1}.

(4) include the straggler in w_t's metric calculation.

Note that (2) requires us to keep an extra window open. (3) requires an updatable statistic.

3.4.3 Update Rules

We can compute aggregates using an update rule. For example, if we want to keep an average, we need to store the sum and the count. Our update rule would then be $\text{sum}_t = \text{sum}_{t-1} + x$ and $\text{count}_t = \text{count}_{t-1} + 1$ then compute $\frac{\text{sum}_t}{\text{count}_t}$.

3.5 Special Aggregations

Some special aggregation tasks include set containment, duplicate removal, and distinct counts.

3.5.1 Bloom Filter

A **bloom filter** is a probabilistic data structure that may return false positives; i.e. $x \in F$ when $x \notin F$ for our bloom filter F. It is defined as a bit string with length m and is associated with k hash functions.

Algorithm: To check if we have already seen an element $x \in S$ for a set S,

(1) compute a hash function (mod) on x: $h_1(x), \ldots, h_k(x)$.

(2) Each hash mod is the position in the array, with k positions total.

(3) Check the bits in each of these positions. If *all* of the bits are 1's, $x \in F$ with probability p and *may* have seen x before. Otherwise, $x \notin F$.

(4) If $x \notin F$, set the bit at each position to 1.

Note: There may be hash collisions, which is why we may return a false positive. We can minimize this by maintaining a low ratio of hash functions to bit length m, as well as controlling the number of entries in the bloom filter.

Example

Suppose we have a bloom filter F with length $m = 10$ and $k = 3$. Our initial filter looks like

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

with all entries 0.

$$h_1(``ucla'') = 2,\ h_2(``ucla'') = 4,\ h_3(``ucla'') = 5, \text{ so } F \text{ becomes}$$

0	1	2	3	4	5	6	7	8	9
0	0	**1**	0	**1**	**1**	0	0	0	0

with no collisions.

$$h_1(``bruins'') = 0,\ h_2(``bruins'') = 6,\ h_3(``bruins'') = 9, \text{ so } F \text{ becomes}$$

0	1	2	3	4	5	6	7	8	9
1	0	1	0	1	1	**1**	0	0	**1**

with no collisions.

$$h_1(``westwood'') = 3,\ h_2(``westwood'') = 5,\ h_3(``westwood'') = 6, \text{ so } F \text{ becomes}$$

0	1	2	3	4	5	6	7	8	9
1	0	1	**1**	1	**1**	**1**	0	0	1

with collisions at 5 and 6.

$$h_1(``wooden'') = 2,\ h_2(``wooden'') = 4,\ h_3(``wooden'') = 8, \text{ so } F \text{ becomes}$$

0	1	2	3	4	5	6	7	8	9
1	0	**1**	1	**1**	1	1	0	**1**	1

with collisions at 2 and 4. There is no false positive since $F[8] \neq 1$.

$$h_1(``boo\_usc'') = 0,\ h_2(``boo\_usc'') = 5,\ h_3(``boo\_usc'') = 6, \text{ so } F \text{ becomes}$$

0	1	2	3	4	5	6	7	8	9
1	0	1	1	1	**1**	**1**	0	1	1

with collisions at 0, 5, and 6. There is a false positive since $F[0] = F[5] = F[6] = 1$.

The probability of a false positive is $p_{false\ positive} = (1 - e^{\frac{kn}{m}})^k$ where k is the number of hash functions, m is the number of bits, and n is the number of entries.

3.5.2 HyperLogLog

A **HyperLogLog** is similar to a bloom filter, but we treat the bitstring as a binary number and use run lengths and probability to provide an estimate of the number of distinct objects in the set. It is spatially efficient since we can count approximately four billion distinct elements using only 5 bits since $\log(\log(2^{32})) = 5$. They have an average error rate of two percent. To use, you would construct a HyperLogLog for each window of messages, and for each key being counted.

Lecture 13

4.1 NoSQL

Definition: NoSQL

NoSQL refers to two concepts:

(1) A database system where the DML and DDL are *not* SQL.

(2) A database system that does not use the relational model. They are usually document-oriented stores, network/graph databases, or key-value stores.

Common models used for NoSQL are:

→ Key-Value stores (Redis)

→ Columnar (Cassandra)

→ Document store (MongoDB)

→ Graph (Neo4j)

4.1.1 The CAP Theorem

Theorem (CAP Theorem)

A distributed system can satisfy at most two out of the three when a network partition or failure occurs.

(1) **Consistency:** Every read receives the most recent write, or an error occurs.

(2) **Availability:** Every request receives some kind of response, not necessarily a correct one.

(3) **Partition tolerance:** The system continues to function even if messages are lost or delayed.

Distributed systems require partition tolerance, so they can be consistent or highly available, *not* both.

4.1.2 Key-Value Stores (Redis)

Redis stores values by keys but can also store data structures. It has interesting data types:

→ Single key-value pairs

→ Hash tables

→ HyperLogLog and Bloom Filters

→ Streams

→ Geospatial (geohash) including radius queries

Redis operates on a *single node* configuration. Therefore, the CAP theorem *does not apply.* Typical use cases include: session caching, message queues, leaderboards, fast lookup/indexing.

Example

If we have a distributed cache, then our system would now be available and partition tolerant, since we are now a distributed system.

4.1.3 Columnar Database (Cassandra)

A columnar database is similar to the relational model, but rows can have different numbers of columns, and adding columns is simple. Given n fixed columns, each row uses a subset of $m \leq n$ columns.

In Cassandra, every node is equal; i.e. there are no masters or workers. This is done by using the **gossip protocol**, making this database **eventually consistent**.

Columnar databases are designed to be denormalized for fast lookups, and can be used as data warehouses. Queries written against Cassandra *cannot* use joins or subqueries, since it isn't relational. There is no concept of a foreign key, but primary keys exist. Rows are also strictly ordered.

CAP Theorem: Cassandra is available and partition tolerant.

4.1.4 Document Store (MongoDB)

A document store takes in full documents of information in a particular format (e.g. JSON/BSON). Documents are schemaless.

In MongoDB, documents (records) are schemaless and stored as a BSON. Records are organized into collections, which are organized into databases; i.e. records $\subsetneq$ collections $\subsetneq$ databases.

Find: The `find` query looks like:

```
db.Books.find(
    { "category": "programming" }, // Where (WHERE)
    { "_id": 0, "title": 1 }       // Projection (SELECT)
    )
    .sort({ "title": 1 })          // Sort (SORT)
    .limit(5);                     // Limit (LIMIT)
```

Aggregation: The `aggregate` query looks like:

```
db.Books.aggregate([
    {
        "$match": { "category": "programming" }  // Match (WHERE)
    },
    // Group (GROUP BY)
    // Note; _id: null for global aggregates.
    {
        "$group": { "_id": "$language", "AvgPrice": { "$avg": "$price" } }
    },
    {
        "$project": { "_id": 1, "AvgPrice": 1 } // Projection (SELECT)
    },
    {
        "$sort": { "AvgPrice": -1 } }           // Sort (SORT)
    },
    { "$limit": 10 }                            // Limit (LIMIT)
]);
```

Differences from RDBMS: There is a lot of redundancy and no concept of normalization. All related data stays together, even if it means duplicating it. There is no concept of a `JOIN` and no schema.

CAP Theorem: MongoDB ensures strong consistency by allowing writes only on the primary replica. The master is the single point of failure. We can improve availability using secondary replica sets or one or more copies of the exact same data. On each write, tehe secondaries must acknowledge the write or else it fails. All reads are done on the primary replica by default. If it fails, we can defer to one of the secondary replicas. Therefore, by default Mongo is consistent and partition tolerant.

4.1.5 Graph Database (Neo4j)

In a graph database, data is stored as either a **node (vertex)** or an **edge**. Nodes represent entities (relations) and edges represent relationships (foreign keys). Nodes and edges can have attributes called **properties**. Each node and edge can also have a **label (type)** associated with it. Edges are usually directed for semantic reasons, but can be traversed in either direction. We an also have self loops.

Neo4j (Cypher) has two common constructs.

Match: `MATCH` specifies a node-relationship-node for pattern matching.

Where: `WHERE` is similar to the SQL `WHERE`, but applies to the properties of nodes.

Example

An example query looks like:

```
    MATCH (gene:Person {name: "Gene Wilder"})-[ACTED_IN]->{movie:Movie}
    WHERE movie.year < $yearParameter
    RETURN movie;
```

with the format: `var:Label { name: "to_match"})-[var:EdgeLabel]->{var:Label}`. In the example above, we look for a node with type Person and name “Gene Wilder”, assigning a temporary name “gene“. Then we extract only the ACTED_IN edges. On the other side, we extract nodes of type Movie and assign a temporary name “movie” to them. Then we return all movies that aired before some $yearParameter.

Neo4j supports, but doesn’t require schemas that define indices and constraints:

- Unique node
- Node property existence
- Relationship property existence
- Constraints aside from uniqueness (pay to win)
- Many data types. Properties can be standard or composite types.

Example

```
// unique node.
CREATE CONSTRAINT constraint_name IF NOT EXISTS ON (n:LabelName)
ASSERT n.propertyName IS UNIQUE;

// node property existence.
CREATE CONSTRAINT constraint_name IF NOT EXISTS ON (n:LabelName)
ASSERT EXISTS (n.propertyName);

// edge property existence.
CREATE CONSTRAINT constraint_name IF NOT EXISTS
ON ()-"["R:RELATIONSHIP_TYPE"]"-()
ASSERT EXISTS (R.propertyName);
```

with the format: `var:Label { name: "to_match"})-[var:EdgeLabel]->{var:Label}`. In the example above, we look for a node with type Person and name “Gene Wilder”, assigning a temporary name “gene“. Then we extract only the ACTED_IN edges. On the other side, we extract nodes of type Movie and assign a temporary name “movie” to them. Then we return all movies that aired before some $yearParameter.

`JOIN`’s are relational but the equivalent in graph databases is traversing a graph.

CAP Theorem: Neo4j is not distributed or sharded, so CAP theorem doesn’t apply. It is not distributed due to clustering.

Lecture 14

5.1 Disk

The storage engine/manager is responsible for managing teh data in memory and on disk. many systems optimize data storage for rows, but some optimize for columns. Some database systems store data entirely in RAM (e.g. Redis), while others use swapping. It is worth noting that disk is *very* slow.

There are various data storage methods:

→ RAM is fast, expensive, random access, and volatile.

→ SSD's are faster than HDD's, aren't as expensive, have fast random access, and are non-volatile.

→ HDD's are cheap, slow, and have bad failure rates.

→ Optical is very slow.

→ Tape is very very slow, but it's cheap and stores data sequentially.

→ Cloud is variable.

5.1.1 Disk Performance

There are various metrics for performance:

→ Access time is the time from a read/write request to block transfer.

→ Seek time is the time from the head to move from parked to a particular sector.

→ Rotational latency is the time it takes to find a sector *once the head is on the track.*

→ Data transfer rate is the time it takes to transfer a sector to RAM.

→ Mean time between failures (MTBF) is the average time between disk failures.

5.1.2 Disk Organization and Access

Data on disk is addressed by a **block number**. Data is transferred between RAM and disk in terms of **blocks (page)**. Accessing a disk can be done sequentially or at random.

Sequential Access: We access blocks in a predetermined order, usually contiguously.

Random Access: Blocks are distributed randomly throughout the disk.

Mitigating Disk I/O: There are several techniques to reduce the disk I/O:

→ Buffering: We can cache n blocks in a buffer.

→ Read-ahead: We can fetch b_{i+j} when we read b_i.

→ Scheduling: We group read/write requests by cylinder.

→ File organization: Organize records/blocks on disk in a particular way.

→ Non-volatile write buffers. See below.

Non-Volatile Write Buffers: If there are a lot of writes taking place, we may have to wait before we can write the block to disk. We can queue it in a Non-Volatile RAM (NVRAM) while we wait. This way, if the power cuts, the NVRAM queue will be emptied onto disk. If NVRAM fills up, the database will block.

5.2 Records into Files

A database maps records → blocks → files. One or more records fit into a block and one or more blocks form a file. We will assume records $\subseteq$ blocks $\subseteq$ files $\subseteq$ relation, where block size is 4KB.

5.2.1 Fixed Blocks

One approach is to store records in a linked list, where records are fixed size. Then indexing becomes $n \cdot \textit{block size}$. There are several issues. If the block size is not a multiple of the record size, we waste space. When we delete records, we have to rearrange pointers. We can remedy this by addding a pointer to the first empty record.

Deletes: Deleting is assumed to be rare in databases, but inserts are not. If we want to insert a record, we look for the block it should belong to and, if there is space, insert. If there is no space, we construct an overflow block.

Overflow Blocks: A file may start out as sequential and contiguous access on disk. However, over time we may need to construct overflow blocks which break contiguity. This is acceptable when overflow blocks are rare, but over time, the performance suffers and so we need to rebuild the data file.

5.2.2 Variable-sized Blocks

We can split the record into two parts: a fixed part and a variable-length part. It looks like the following: A table of contents points to the variable-sized attributes followed by the data for the fixed sized attributes, a null bitmap that partitions the table of contents and fixed data from the variable-length data, and the data for the variale-length attributes. We store a pointer to each record in the file header. The format for the table of contents is `(start_byte, length)`, and are 4 bytes wide. The null bitmap indicates which fields are null.

5.2.3 Records into Blocks

Records are stored in blocks. Each block contains a header with: the number of records in the block, a pointer to the end of the free space in the block, and the location and size of each record.

Aside

Given the previous discussion, `ALTER TABLE` is a bad idea since it changes the format and size of the record. After modification, the record now needs to store extra information, which may lead to the creation of overflow blocks since the size of each record may have grown and may exhaust free space.

Lecture 15

6.1 Files to Indices

Since records were stored sequentially, finding a specific record by key requires scanning the full file. So, lookup takes $\mathcal{O}(B)$ block transfers for B blocks, with an average of $\frac{B}{2}$. This would be required for every `JOIN`, `WHERE`, etc.

6.2 Indexing

An index works as follows: A search key (value) is put into an indexing function, which returns the block address and byte offset of the particular record(s). There are two types of indices:

→ Ordered indices are based on a sorted ordering of the values of the sort-key. It requires sequential (maybe contiguous) access.

→ Hash indices are based on a uniform distribution of values across a range of buckets. This is random access.

An index is usually too large to fit into RAM, so it must be stored on disk. Therefore, we get the best performance when we read the index file sequentially. If the entire index fits in RAM, the key lookups in the index are trivial. This still requires us to read it into RAM.

6.3 Index Sequential Access Methods (ISAM)

Ordered indices can be (1) primary (clustering) or (2) secondary (non-clustering) and (1) dense or (2) sparse.

6.3.1 Primary/Clustering Index

In a **primary/clustering index**, we pick a search key. The keys in both the index and records are sorted by the search key. This dictates the order of the underlying data file. It *does not* need to be a primary or unique key, though it usually is. There is at most *one* primary index on a table. The primary index returns a block number and a byte offset that points to the specific record in the file. If there are duplicates, we point only to the first matching record.

Dense: If the index is **dense**, there is an index for every record.

Sparse: If the index is **sparse**, we only keep $k < n$ indices for n records. When querying, we have to search for the largest index that is less than the requested record and do a linear search. A sparse index implies that we may be able to read the entire index into RAM. To minimize block transfers, we may want to add the first search-key in each block into our index.

Pros and Cons: Primary indices are good for range queries and minimal block reads. However, it is difficult to maintain contiguity on disk because inserting a record into a full block requires creating overflow blocks. Over time, the file needs to be rebuilt to restore contiguity.

6.3.2 Secondary/Non-Clustering Index

In a **secondary/non-clustering index**, the data file is sorted by some search key different from the one used to create the index. Secondary indices can *only* be dense. This is because the order of the indices are different from the order of the data. For duplicate keys, secondary indices must point to both records.

6.4 B$^+$-Trees

Definition: B$^+$-Tree

A **B$^+$-Tree** adds an extra layer atop an ordered index that points to individual keys. The root and interior nodes form a sparse index, while the leaves form a dense index and contain the keys that *point to* records. Each level represents one disk seek and block transfer.

Under a purely sequential index, data is initially stored contiguously, but may devolve to random access due to overflows. Eventually, it degrades the performance of the index and we may have to rebuild the index and data file. By organizing the index into a B$^+$-tree, we can guarantee high record-search performance.

Full Table Scan and Full Index Scan: In a full table scan, we transfer blocks to search for records by following record pointers. But in a full index scan, we transfer blocks in order to search for the keys, then do an additional search for the actual record.

6.4.1 Notation

The **height** of the tree h is defined to be the number of nodes from root to leaf, *counting both* (counting fence posts). The **left pointers** are *strictly less than* ($<$) and **right pointers** are *greater than or equal to* ($\geq$) the node value.

Leaf Nodes: A leaf node contains (up to) n pointers and (up to) $n-1$ key-values. For a B$^+$-Tree to maintain balance, a leaf node must contain at least $\left\lceil \frac{n-1}{2} \right\rceil$ and at most $n-1$ key-values. Leaves form a dense index. As such, each leaf points to the next leaf node.

Root and Internal Nodes: A root and internal node contains at least $\left\lceil \frac{n}{2} \right\rceil$ pointers and (up to) n pointers.

6.4.2 Time and Space Complexities

Insertion and deletion are both $\mathcal{O}\left(\log_{\left\lceil \frac{n}{2} \right\rceil} K\right)$ for K keys and an n-ary tree. There is also additional space overhead from the internal nodes. There may also be wasted space since nodes need only be half full. However, it is more efficient than rebuilding the index.

6.4.3 Disk I/O Costs

Action	Cost
Traversing between levels	$t_S + t_T$
Traversing between leaves	t_T
Fetching a record	$t_S + t_T$

where t_S is one block seek and t_T is one block transfer.

6.4.4 Search

Use your eyes lol. The disk complexity to search for *one* key is $h(t_S + t_T)$ operations.

Range Queries: For $\{k : k \in [i, j]\}$, we search for i and collect record pointers until we reach j. For an *inclusive* range query, we do $h\left(t_S + t_T\right) + \left(\left\lceil \frac{j-i+1}{n} \right\rceil\right) t_T$ operations, where n is the branching factor.

6.4.5 Insertion

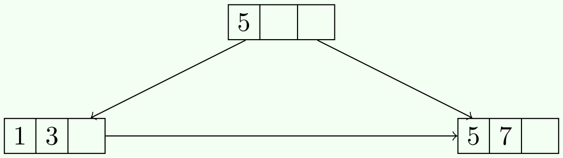
We show insertion through example:

Example

Suppose $n = 4$. Then we have at most $n - 1$ entries for a node. We will insert $\{1, 3, 5, 7\}$. After inserting $1, 3, 5$, we get

1	3	5

When inserting 7, note that it will overflow the root node. So we split at $\left\lceil \frac{n}{2} \right\rceil$ and create a new parent node to get



we do this recursively. We always insert into the leaf nodes and propagate up.

6.4.6 Indexes in SQL

```
CREATE INDEX name_of_index ON table
[ USING (btree | hash | gist | spgist | gin | brin) ]
(
    attr1 [ASC | DESC] [NULLS { FIRST | LAST }],
    ...
);
```

by default, creates a B$^+$-tree on `attr1`. A primary key automatically defines an index.

6.5 Hash Indices

Hash indices are good at equality queries, but not good at range queries. Each key k_i points to a bucket b_i via a hash function $h : K \to B$. This bucket points to a linked list of records in RAM, or blocks on disk matching key $h(k_i)$.

6.5.1 Search

We compute $h(k_i)$ to get bucket b_i and iterate through the blocks/records to find the one we are looking for. We might have to iterate through due to hash collisions.

6.5.2 Deletion

To delete a key, compute $h(k_i)$ and find all records matching k_i and delete them from the block/list of records.

6.5.3 Bucket Overflow

We might have **insufficient buckets**, so it devolves into a linear search. Another issue we may have is **bucket skew**, where many records hash to the same bucket, leading to a non-uniform distribution. One formula for determining the number of buckets is $|B| = 1.2 \cdot \frac{n_r}{f}$ where n_r is the number of records and f is records per bucket.

Lecture 16

7.1 Spatial Indexing

If we want to extract other points that are within a certain distance of a user, we can use a **nearest neighbor query**. In one dimension, we simply divide and conquer.

7.1.1 *kd*-Trees and Ball Trees

In higher dimensions we can use a ***kd*-tree**. We partition our search space, and recursively perform a n-ary search over the space remaining. Note that a *kd*-tree will not center you. Rather, it finds all points in a particular search space. A **ball tree** uses a radius instead of axes.

Example

If we want to find all points relative to a particular location, we filter based on the desired conditions. Then, we query the location of the person in the *kd*-tree and return the results within a certain boundary (polygon).

7.2 Query Plans

There are many ways to write queries. The optimizer will convert a query into an efficient **query plan** that is invisible to the user. It is similar to compiling code into byte code. Query processing has three steps:

(1) The query gets parsed and translated into relational algebra.

(2) We perform a series of query-optimizing transformations.

(3) We evaluate the query using statistics.

Since (1) is trivial, we start at (2).

Example

Different queries may lead to the exact same result set. For example,

```
    SELECT a
    FROM r
    WHERE c > 1000;
```

can be written as either

(1) $\sigma_{c>1000}\left(\Pi_{a,c}\left(r\right)\right)$

(2) $\Pi_{a,c}\left(\sigma_{c>1000}\left(r\right)\right)$

After we parse the query into their relational algebra equivalent(s), we determine the "better" query. However, the RDBMS needs to know a variety of things to estimate the better query. Some of these metrics include:

(1) Estimated block transfers and seek.

(2) Number of tuples.

(3) Current CPU/RAM state[1].

(4) Data or Network transfer speed.

(5) Disk space.

(6) Time.

7.2.1 Estimating Cost

For data operation, we compute the execution time or the block reads and disk seeks; i.e. disk I/O.

Note: In this class, the disk seek is defined as the time it takes for the head of the disk to find a particular sector *from a parked state.* It is *not* a seek if you're already on the same track.

in our computations, t_T will be the time it takes to transfer one block from disk to memory, and t_S the average block access time (seek time + rotational latency). The amount of time to transfer B blocks with S random accesses is then

$$Bt_T + St_S$$

7.3 Select

Given a `SELECT * FROM r WHERE` Ψ, we can naively perform a full table scan which reads every single record in a file when there is no primary key. This becomes $\mathcal{O}(n)$ time for n records, or $\mathcal{O}(B)$ time for B blocks.

In the following examples, let b_r be the number of blocks in the file, t_T the time it takes to transfer one block, and t_S the seek time.

Example

Suppose we have a sequental file *without an index*, and we want to find records that match on some *non-key* attribute. The total execution time is

$$t_S + b_r \cdot t_T$$

Notice that we seek to the start of the file before performing (up to) $b_r \cdot t_T$ block transfers.

If the blocks are not stored contiguously, then in the *worst case*, we seek for every block to get

$$b_r(t_T + t_S)$$

[1] Postgres uses this one.

Example

If we search for a *key*, then the worst case is

$$t_S + b_R \cdot t_T$$

and happens when the record either DNE or is in the last block. The best case is

$$t_S + t_T$$

and happens when the record is in the first block. The average time is

$$t_S + \frac{b_r \cdot t_T}{2}$$

If we index by a primay B$^+$-tree, then traversing from root to leaf is

$$h(t_S + t_T)$$

Searching for a record pointer in the block requires 0 disk I/O since the block is read into RAM. Retrieving the associated key is

$$t_S + t_T$$

so the total time spent on disk I/O is

$$h(t_S + t_T) + t_S + t_T = (h + 1)(t_S + t_T)$$

Range Queries: Suppose we want to perform a query on v such that $k > v$. Then, we want to find v which takes

$$h(t_S + t_T) + t_S$$

time to get the first record. Fetching all records after takes bt_T for b remaining blocks *at worst*.

$$h(t_S + t_T) + t_S + bt_T$$

To get $k \leq v$, we can just process the records starting from the beginning via full index scan *instead of* a B$^+$-tree until we get $k = v$. This means it takes

$$t_S + bt_T$$

for b remaining blocks.

7.4 Joins

7.4.1 Nested-Loop Join

A **nested-loop join** is a brute force way to compute $R \bowtie_\theta S$. The algorithm is

```
result = {}
for each tuple $t_r \in R$:
    for each tuple $t_s \in S$:
        if pair $(t_r, t_s)$ satisfies $\theta$:
            result = result $\cup (t_r, t_s)$
return result
```

where the left-hand side is the **outer** relation and the right-hand side is the **inner** relation. This algorithm does not require an index, and thus is very slow. There are also no restrictions on θ. If we have n_R tuples in R and n_S tuples in S, we get $\mathcal{O}(n_R \cdot n_S)$ time complexity.

Example

Suppose we have $n_R \cdot n_S$ pairs of tuples. For every record in R, we have to do a full table scan of S. Let b_R and b_S be the number of blocks in R and S respectively. Then the total number of seeks and block transfers are

$$b_R + n_R$$

and

$$n_R \cdot b_S + b_R$$

respectively.

Example

Suppose $R \bowtie S = S \bowtie R$. If one relation fits entirely in memory, then its blocks are read in exactly once. If R fits entirely into memory, then we perform

$$1 + 1 = 2$$

block seeks and

$$b_R + b_S$$

block transfers if R is the **inner** relation and

$$1 + n_R$$

block seeks and

$$b_R + n_R \cdot b_S$$

block transfers if R is the **outer** relation.

7.4.2 Block Nested-Loop Join

A **block nested-loop join** is a nested-loop join but we process the outer relation by *block* rather than by tuple. The algorithm is

```
result = {}
for each block $B_R \in R$:
    for each block $B_S \in S$:
        for each tuple $t_r \in R$:
            for each tuple $t_s \in S$:
                if pair $(t_r, t_s)$ satisfies $\theta$:
                    result = result $\cup (t_r, t_s)$
return result
```

For each block in R, we load it, so we get b_R blocks. we load b_S blocks for each block in R to get

$$b_R + b_r \cdot b_S = b_R(b_S + 1)$$

block transfers and

$$2br$$

block seeks.

7.4.3 Indexed Nested-Loop Join

An **indexed nested-loop join** is a nested-loop join but we put an index (using a B$^+$-tree or hash) on the inner relation. The algorithm is: For each tuple $t_R \in R$, look up the key from $t_R \in S$ and retrieve all matching tuples to perform a join. This way, once we have an index for S, we only need to scan R to perform the join. Scanning R takes

$$b_R(t_T + t_S)$$

time and indexing takes

$$cn_R$$

where c is the disk cost of looking up a record from an index. Thus, the total time is

$$b_R(t_T + t_S) + cn_R$$

in a B$^+$-tree, $c = (h + 1)(t_S + t_T)$, so we get

$$b_R(t_T + t_S) + n_R(h + 1)(t_S + t_T)$$

7.4.4 Merge Join

An **merge join** sorts both relation before joining them. We sort on $R \cap S$. This becomes an interleaved linear scan since we only scan each relation once. Then the number of block seeks is

$$\left\lceil \frac{b_R}{b_B} + \frac{b_S}{b_B} \right\rceil$$

and the number of block transfers is

$$b_R + b_S$$

where b_B is the buffer size.

7.4.5 Hash Join

An **hash join** has a **build side** and a **probe side**. The build side loads as many blocks w that fit into RAM, building a hash table on it. Then we do a full table scan over S and check if the hash is in S. Then we load in the next w blocks, repeating this process. We do $|w|$ windows of blocks worth of full table scans on S. We then post-filter since we might have hash collisions.

7.5 Joins in Spark

Spark supports several join algorithms. There are two (orthogonal) types of join algorithms:

- → Broadcast Hash/Nested-Loop Join
- → Shuffle Hash/Sort-Merge Join

7.5.1 Broadcast Joins

If one of the tables (e.g. R) can fit into RAM on every executor and the driver, the table is **broadcasted** to all executors in the cluster. We then partition the bigger table into n partitions, joining them on R in parallel. This is *not* good for outer joins since we cannot perform these in parallel due to joining with `NULL`'s on no match by default.

```
from pyspark.sql.functions import broadcast
...
large_df.join(broadcast(small_df), ["join_key"])
```

or

```
SELECT /*+ BROADCASTJOIN(small) */ *
FROM large JOIN small
ON large.foo = small.foo
```