Contents Page

Week 1 - Introduction

Key Principles

- Data Independence: Application decoupled from structure of data.
- Declarative Interface: Specify what rather than how.
- Space can be reused but not time: Speed up lookups and joins using indexing and copies.
- Scale-agnostic design: Doesn't matter if parallel nodes or just one node.

Definition (DBMS). Software package manager that manages a database. Performs the tasks of

- Store data/ backup/ recovery
- Supports high level access language such as SQL
- Application describes database accesses using that language
- DBMS interprets statements of language to perform requested database access

Levels of abstraction

1. View: Describes how user sees the data
2. Conceptual schema: Defines **logical** structure of data
3. Physical schema: Defines the file and indexes used.

Data Independence

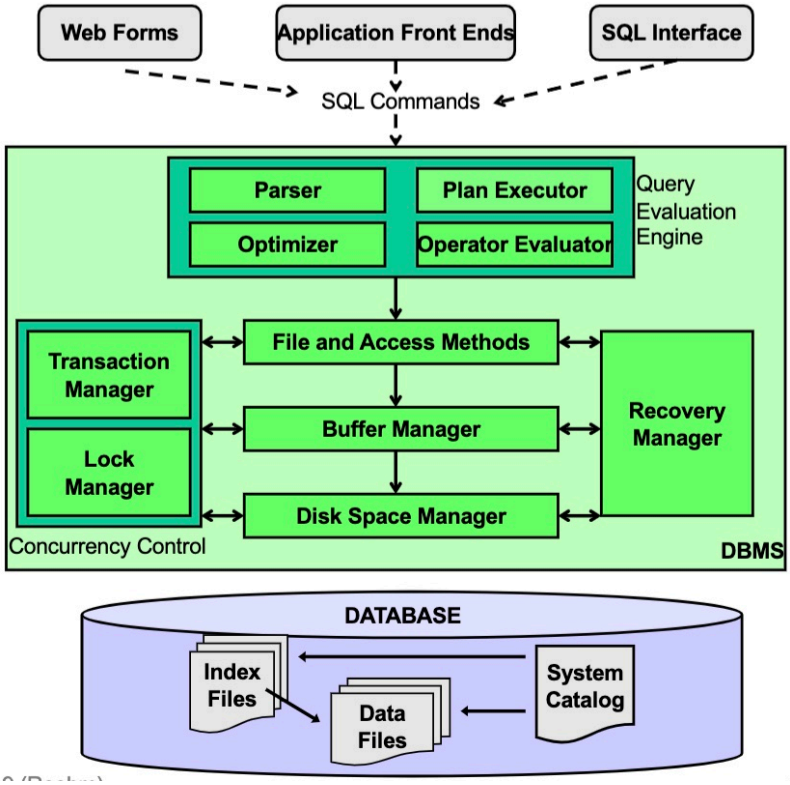
We want applications to be insulated from how data is structured and stored.

1. Logical Data Independence: Protection from changes in logical structure of data
2. Protection from changes in physical structure of data

Query Processing

When we run a SQL query, we have 3 things to do in order to process the query

1. What is the internal representation of the query?
2. How can we optimise the query?
3. How do we actually execute the query?

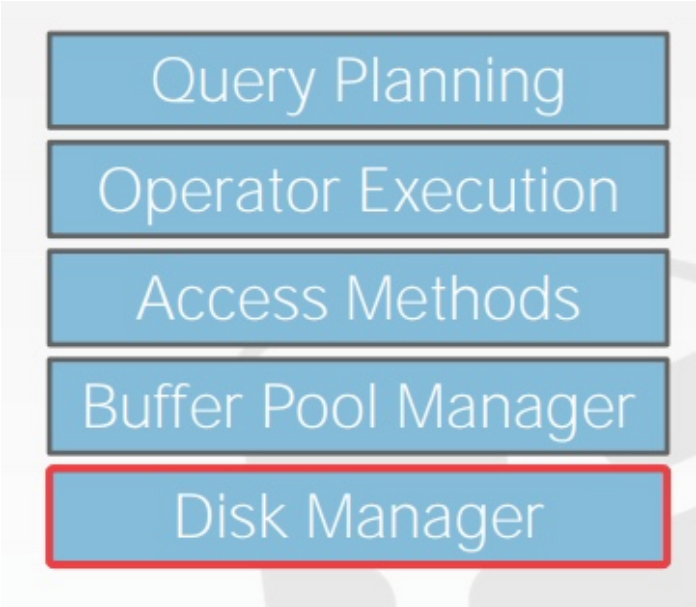

Web Forms
Application Front Ends
SQL Interface
SQL Commands
Parser
Plan Executor
Optimizer
Operator Evaluator
Query Evaluation Engine
Transaction Manager
Lock Manager
File and Access Methods
Buffer Manager
Disk Space Manager
Recovery Manager
Concurrency Control
DBMS

DATABASE
Index Files
Data Files
System Catalog

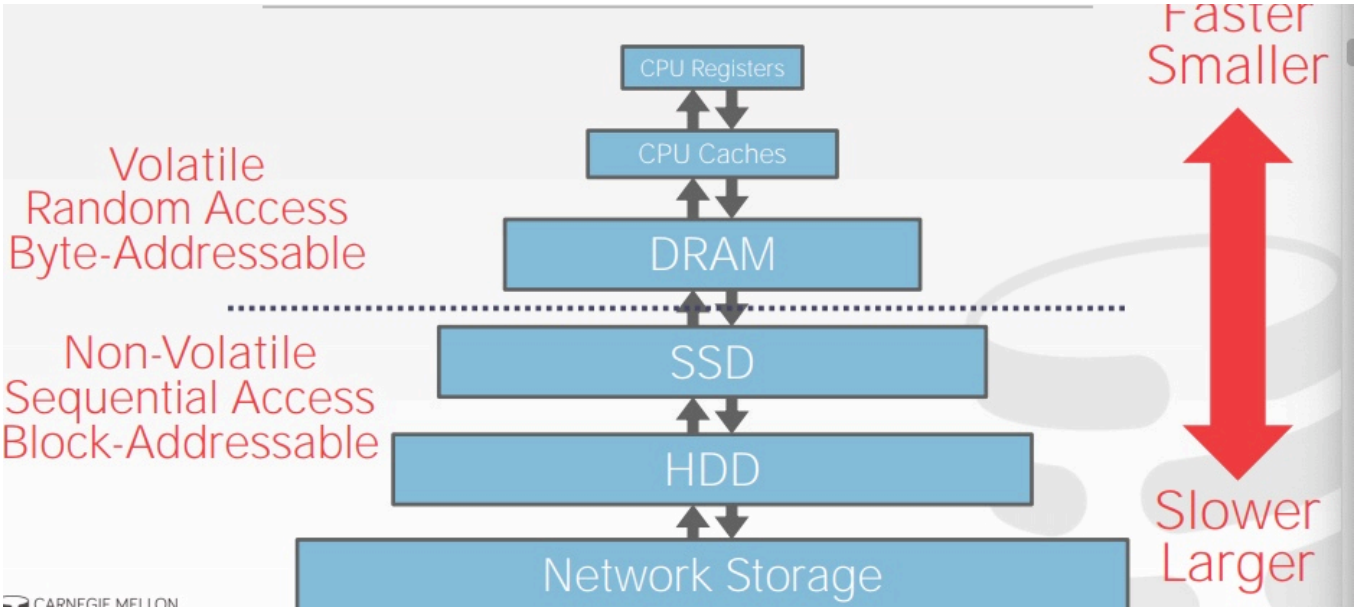
Week 2 - Storage Layer and Buffer Management

DBMS Storage Hierarchy

The DBMS structure looks like this.



In particular, we have the memory hierarchy



CPU processes the data stored in RAM. However, RAM is **volatile**, meaning you lose the data if you turn the power off. Furthermore, the amount of memory in RAM isn't so large. Reading and writing from CPU to RAM is quite a fast process.

However, secondary storage (HDD) is cheap, stable, and can store large amounts of data. However, reading and

writing from RAM to HDD is an expensive operation and hence should be done intelligently. The reason that secondary storage is slow is that it is physically storing data on the drive and when you are retrieving data, you need to find the physical location of the data.

Definition (Sequential acess). A group of elements (such as data in a memory array or a disk file or on magnetic tape data storage) is accessed in a predetermined, ordered sequence.

Definition (Random access). The ability to access an arbitrary element of a sequence in equal time or any item of data from a population of addressable elements roughly as easily and efficiently as any other, no matter how many elements may be in the set.

Remark Random access on an HDD is slower than sequential access. Traditional DBMS systems try to maximize sequential access such that it will try to write data in contiguous blocks.

Alot of features in the DBMS may seem similar to the OS, however, the DBMS outperforms the OS for:

1. Flushing dirty pages in correct order
2. Buffer replacement policies
3. Thread/process scheduling
4. Prefetching

Disk Storage Organisation

We now look at how the DBMS represents the database in files on the disk. There are 3 topics to consider from the ground up

1. Tuple Layout
2. Page Layout
3. File Storage

Tuple Layout

Definition (NULL Bitmap). A null bitmap just tells us whether a column record has a value of NULL or not. This is useful because whilst for variable length columns, we can just check the size == 0 to see if it is NULL, for fixed length columns, this can be difficult.

Definition (Tuple). A tuple is a sequence of bytes where the DBMS needs to interpret those bytes into attribute types and values. There are two parts to a tuple, the **tuple header** and the **tuple data**. The tuple header contains meta-data about the tuple such as a NULL bitmap, it does not include meta-data about the schema of the DB!. The tuple data are attributes are stored in the order that you specified when you specified the schema when creating the table.

Defintion (Tuple and Record). A tuple refers to the logical level (think pandas dataframe) whilst the record refers to the physical level (think of c-strings).

Definition (Record ID). The DBMS keeps track of individual tuple via an *unique record identifier* which is **page_id + offset/slot**.

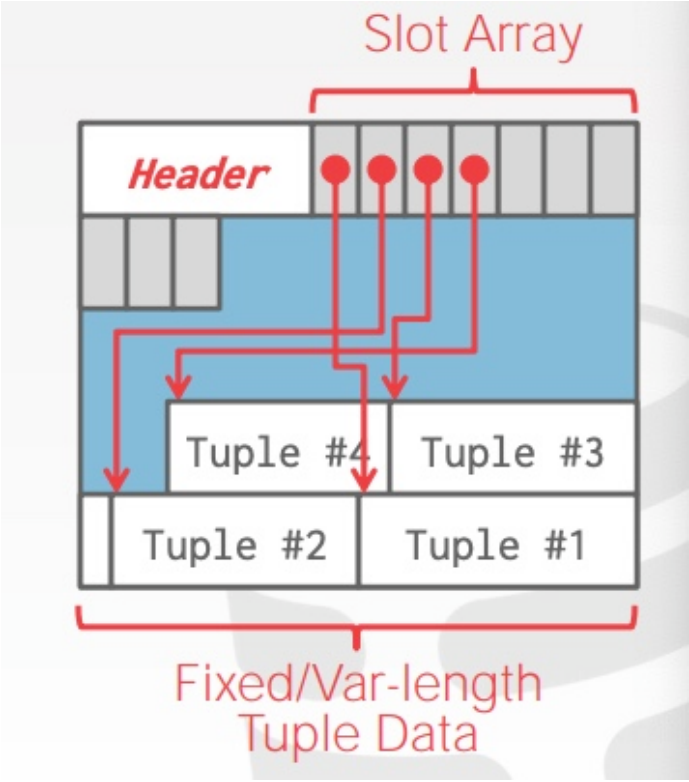
Page Layout

Every page contains a *header* of meta-data regarding page's content such as the page size. We look at tuple-

oriented methods of storing tuples on pages.

Definition (Strawman Approach). We keep track of the number of tuples in a page and then just append a new tuple at the end. However, this is difficult when we have a *variable-length* attribute or when we need to delete tuples.

Defintion (Slotted Pages). The *slot array* maps slots of the tuples starting offset. The header keeps track of the number of used slots and the offset of starting location of **last used** slot. This is the most common approach in DBMSs today.



File Storage

Definition (File). A file is a sequence of records stored in binary format.

The DBMS stores a database as one or more files on disk.

Definition (Storage Manager). The storage manager is responsible for maintaining a database's files. The storage manager is responsible for all aspects of storage, transactional requirements, low-level concurrency, and indexing. It includes modules such as the buffer manager and the lock manager.

Definition (Page). A page is a **fixed-size** block of data. It contains tuples, meta-data, indexes, and log records.

Files are organized as collection of pages. Each page is given an unique identifier in order to map page ids to physical locations. There are 3 ways DBMS can manage pages

1. **Heap File Organization**: Unordered collection of pages where tuples are stored in random order. We can use a **linked list** or a **page directory** to help represent this and keep track of what pages exist and which ones have free space.
   a) **Linked List**: We maintain a *header page* at the beginning of the file that stores two pointers for two linked lists, one to the HEAD of the *free page list* and one to the HEAD of the *data page list*. Additionally, each page keeps track of number of free slots in itself.
   b) **Page Directory**: We have a directory that maps to each page and number of free slots per page.

2. **Sequential/Sorted File Organization**: Store pages in sequential order based on search key. We can use *binary search* to locate records.
3. **Indexes File Organization**: Organise records based on certain search keys.

Row vs Column Store Concept

We looked at storing data in rows, however, it may be more efficient to store data as columns. This is similar to storing data as pandas dataframes for those familiar.

For analytical workloads, column-oriented stores can work better. By organising data by attributes, we can easily query attributes we want.

Advantages of column store

- We don't need to read in non-needed attributes
- Better compression
- Better parallel processing

Disadvantages of column store

- Hard to manage updates
- Annoying when working with full-row access
- Small-table data

Buffer Management in a DBMS

Definition (Lock). Protects the database **logical** content from other transactions. It is held for the transaction duration.

Definition (Latch). Protects the database **physical** data structure from other threads. It is help for the operation duration.

Definition (Buffer Pool). The buffer pool is an in-memory cache of pages read from disk. It is organized as an array of fixed size pages. Each array entry is called a **frame**.

When the DBMS requests a page, an exact **copy** is placed into one of these frames. The buffer manager assists the DBMS for when it needs a page from the disk. The buffer manager is responsble for moving physical pages back and forth from main memory to disk.

The buffer pool keeps the following metadata maintained:

- **Page Table**: In-memory hash table to keep track of pages currently in memory.
- **Dirty Flag**: Get sets when a thread modifies a page and we will need to write it back to disk.
- **Pin Counter**: Number of threads touching that page.

```
class BufferManager:
  def get_page(self, pid):
    """
    Access the page needed.
```

```
    Return: Frame the DBMS would like.
    """
    frame_result = search_buffer_for_frame(pid)

    if(frame_result == None):
      # Frame is not in buffer.
      frame_result = get_next_empty_frame()
      if(frame_result == None):
        # We ran out of space in the buffer.
        frame_result = replacement_policy.choose() # choose which frame to evict.
        if frame_result.is_dirty() == True:
          write_page(frame_result)
          frame_result.dirty = False

        load(pid, frame_result)

      frame_result.pin += 1 # we pin the page we are working with.

      return frame_result
```

Buffer Replacement Policies

A replacement policy is an algorithm that the DBMS implements that makes the decision on which page to evict the buffer when it needs space.

Most Recently Used

Maintain a timestamp of when each page was last accessed. Evict the page with the *youngest* timestamp.

Least Recently Used

Maintain a timestamp of when each page was last accessed. Evict the page with the *oldest* timestamp.

CLOCK

Each page has a reference bit. When a page is accessed, set it to 1. We then organize the pages in a circular buffer with a *clock hand*. Upon sweeping, check if reference bit is 1. If it is, then set to 0. If not, then evict it and increment the hand. If we are reading in a page that is already in the buffer, **do not** increment the hand but do set the reference bit of the page in buffer to 1.

GCLOCK

GLOCK is the same as CLOCK except the reference bit can take on values greater than 1 and when we sweep, we just decrement the count until it reaches 0. When we read in a page that is already in the buffer, we just increment the reference bit. If the page is not in the buffer and the buffer is full, keep iterating the hand and decrementing buffer frames until we find one that is unpinned and has a reference bit of 0. Then increment the hand.

Prefetching

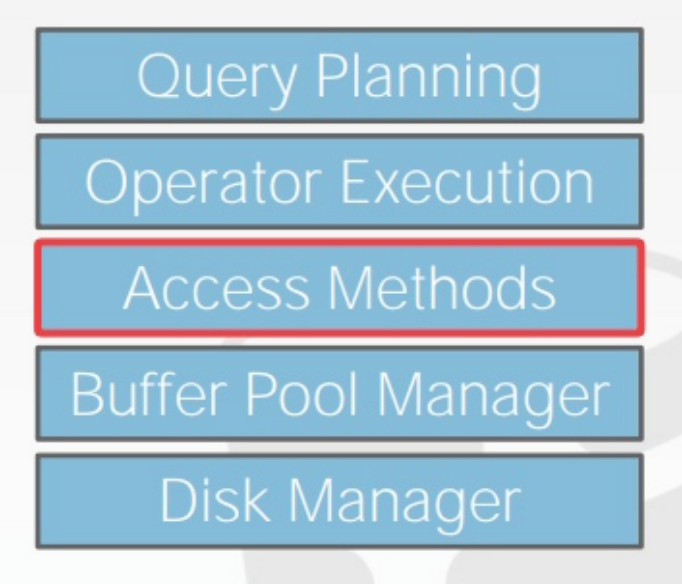
If we know ahead of time what the requests are, we can load them up and pre-fetch them. This is for when we do things like sequential scans of tables or fetching BLOBs which may span across several pages.

DBMS vs OS File System

The DBMS can manage requirements for DBMS alot better than OS.

Week 3 - Tree-Based Indexing

What are Indexes



The access methods refer to how does the database actually access the tuples of interest. These access methods describe how does the database access the tuples without having to resort to **sequential scans**.

This and next week, we will be looking at tree and hashmap data structures. These structures can be used throughout numerous parts in the DBMS such as

- Internal Meta-Data: Keep track of where pages are on disk, who holds what locks etc
- Core Data Storage: The overarching data structure to store tuples
- Temporary Data Structure: These DBMS build this on the fly for its queries
- Table Indexes: Useful to quickly identify tuples for when we execute queries

Sometimes decision representation for table indexes may be bad for temporary data structures due to trade off between space and speed.

We now look at how to support the DBMS execution engine to read and write data from pages using access methods. There are 2 types of data structures that can help us increase the speed of accessing data: *Hash tables* and *Trees*.

Tree data structures are common in table indexes and whilst hashmaps are used in internal meta data. This relates to the fact that trees are order preserving.

Definition (Table Index). A table index is a replica of a subset of a table's columns that are organized and sorted for efficient access using a subset of those columns.

Definition (Ordered Array Index). An array sorted on a search value with a pointer to where the record is stored

on the page. For example, if we had an index on `age`, then the array would be sorted by `age` with pointers to the corresponding entry.

The DBMS *ensures that the content of the table and the index are logically in sync*. It is the DBMS's job to figure out which index is the best to execute each query which we will see more in the query optimization lecture .

There is a trade-off on the storage/maintenance overhead and number of indexes to create per database. Furthermore, whilst having indexes can increase read time, it will slow down insert and delete time as we now need ot update the index to ensure the content of the table and index are in sync.

Tree-Based Indexes

There are 3 types of indexes:

1. Tree Based Indexes
2. Hash Based Indexes
3. Bitmap Indexes

Index Sequential Access Method (ISAM)

This is a **static** index structure, that is, the structure will not change. This is good for equality and range searches.

Definition (Separator). The nodes in the tree used for ISAM. This is comprised of multiple separator entries.

Definition (Separator entry). A tuple (k_i, p_i) where k_i is a search key value and p_i is a pointer to a lower level separator.

Hence, the tree will comprise of multiple separators at each level where each separator entry will point to separators on the lower level of the tree. At the **leaf node**, there are no pointers, just the entry of interest.

The name ISAM comes from the fact that by the time we reach the leaf node, which is an array containing multiple values satisfying the separator conditions, we do a sequential scan to look for tuples satisfying our query.

Contents of the leaf page can change. Deletion and insertion occurs in the leaf nodes. However, row insertion can lead to an overflow in the leaf node. As a result, you can construct a overflow bucket to store the latest entries. Note that this overflow chain will be unsorted and will not necessarily be contiguous to the rest of the tree in memory. Hence, ISAM can be inefficient if we have numerous inserts and deletes occurring.

ISAM are suitable for range and equality searches and for databases that do not require numerous dynamic changes from insertion and deletion operations.

B+ Tree

B Tree

We first look at B trees. A B tree is known as a **balanced M-way tree**, where `M` is the number of nodes does each nodes it points to and we will have `M-1` search keys. By definition, we require that

1. The height of the tree must be kept minimum
2. No empty subtrees above the leaves of the tree
3. Leaves of the tree must be at the same level
4. All nodes except the leaves must have at least some minimum number of children (2).

Lemma (Properties of B tree of order M). We require that the B tree of order M has the following properties

1. Each node has a maximum of `M+1` children and minimum of `M/(2-1)` children.
2. Keys are arranged in order where all keys in the subtree to the left of a key are **predecessors** of the key and on the right are **successors** of the key.
3. When a key is to be inserted into a full node (M-1) keys, then we split the node into 2 notes and the key with the `median` value is inserted into the `parent` node. If the parent node is the root, then we create a new root.
4. All leaves are on the same level with no empty subtrees above the leaf nodes.

Remark The keys in each node must be *ordered*.

Remark In the real world, there is a specific called a B Tree but also people could also use the term to refer to the general family of data structures which include B trees and B+ trees.

B+ Trees

Now a B+ tree builds off a B tree.

Definition (B+ Tree). A B+ tree is a **self-balancing** tree that keeps data sorted and allows searches, sequential access, insertions, and deletions in `O(logn)`.

Remark This differs to a binary search tree as the node can have **more** than two children at each node. Furthermore, they are optimized for sequential access as they can be stored nicely on disk for historical reasons.

Lemma (Properties of B+ Tree). A B+ tree is a M-way search tree with the following properties

1. It is perfectly balanced (every leaf node is at same depth). So every value we look at, it has the same distance to the root which reduces costs.
2. Every inner node other than the root is **at least** half full: $M/(2-1) \leq \#\ keys \leq M-1$
3. Every inner node with k keys has k+1 non-null children.
4. Each leaf node holds between M and 2M index entries.
5. Each index node has between M and 2M+1 children.

Similar to ISAM, B+ trees support equality and range searches, and multiple attribute key and partial key searches. Unlike ISAM, this is a dynamic index structure where we can now modify the data structure to accomodates insertions and deletes in the table.

It is a similar structure as the ISAM tree. In each node, we have `k` key values and `k+1` pointers pointing to other nodes on lower level. Note that the each node is a **sparse** index as we only point to one entry in the next node rather than **all** the entries. This is because we can just follow the single pointer to the next node and then use sequential scan to find the next key of interest and pointer of interest.

On the leaf node, we now have **sibling pointers** where the leaf node will point to the next leaf node. This is known as a **sequence set**. This saves us time for range queries as we can now move along the leaves rather than backtracking up the tree. The sequence set is a **sorted** linked list of index entries.

Definition (Leaf Node Values). We have two options for leaf node:

1. Record ID: A pointer to location of tuple that the index entry corresponds to. Used in PostGres.
2. Tuple Data: The actual content of the tuple is stored in the leaf node. However this isn't feasible if we have multiple indexes. We just do this for the primary key. So we instead then require secondary indexes to store the record id as their values and then go to the other tree indexed by primary key to find the actual value. Used in SQLite and MySQL.

Insertion There are 2 cases to consider:

1. Leaf Node Split: Place first $\lceil n/2 \rceil$ nodes in original node and remaining in new node. Take the value in the new node and also place it in the parent.
2. Index node: Same as leaf node except now you take the middile elements in your original node and **push** it up the tree.

Properties (Height of B+ Tree). The height of a B+ tree is

$$\lceil log_{children}(\frac{numrecords}{leafpageindexes}) \rceil$$

Then we need to add 1 for the root.

Difference between B Tree and B+ Tree

The original B Tree stores keys and values in all nodes in the tree. In the B+ tree, the real value and key is stored in leaf. The issue is that for B tree, deleting a value in leaf, the value could still exist in the inner node somewhere. However, B tree is more space efficient since each key only appears once in the tree whereas in B+ tree, the key can appear multiple times.

Difference between B Tree and ISAM

- Tree is balanced
- Sibling pointers between leaf nodes

Multi-attribute B+ Trees

When constructing multi-attribute B+ tree indexes

1. All attributes are in the suffix of the key (so things we aren't indexing on must come last)
2. Any inequality conditions must be the last attribute (but (1) overrides this as (1) will not affect our query)

How to effectively use indexes in physical database design

Clustered vs Unclustered Index

Definition (Clustered Index). Rows are stored physically on the disk in the **same order** as the index. There can only be one clustered index. This benefits from sequential access.

Definition (Unclustered Index). There is a second list that has points to physical rows. You can have multiple non clustered indexes. However, with each new unclustered index, it will increase the time to update and delete records.

Remark Clustered index is faster than an unclustered index if searching off the same key since you can go directly to the physical data. However, updating the table with a clustered index will take longer due to having to physically move the rows around.

Dense vs Spare Index

Definition (Dense Index). An index record appears for every search key value in the file. This record contains

search key value and a pointer to the actual record.

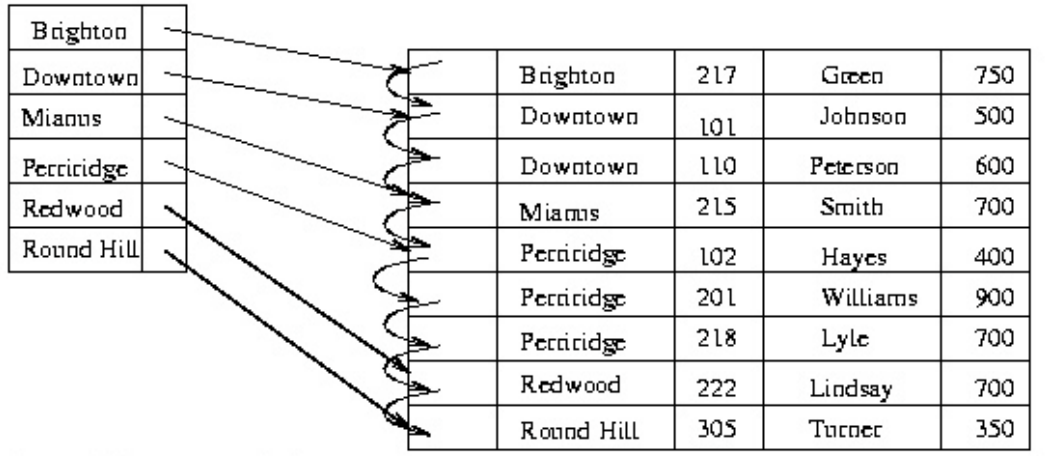


Figure 11.2: Dense index.

Definition (Sparse Index). Index records are created only for **some** of the records. To locate a record, we find the index record `L` with the **largest** search key value less than or equal to the search value we are looking for. We then sequentially iterate through the records from the one pointed by `L` until we find the desired record.

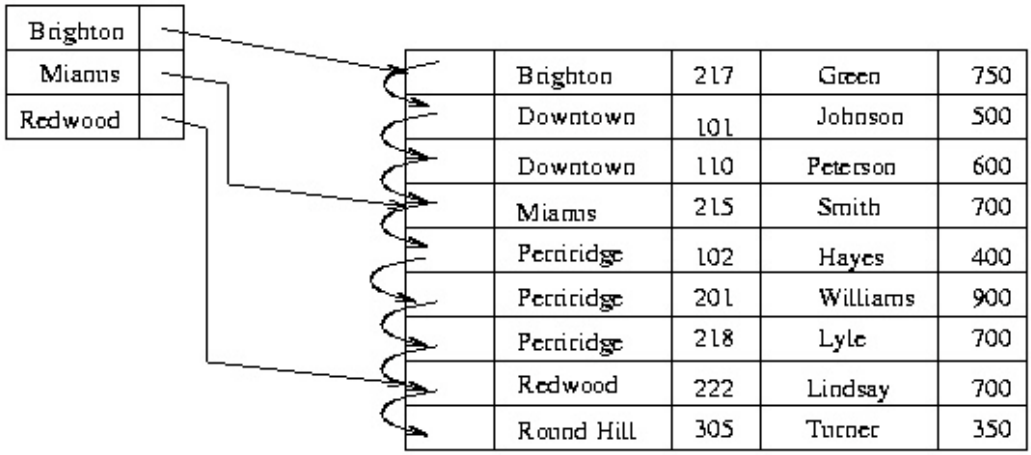


Figure 11.3: Sparse index.

Remark Dense indexes are faster but sparse indexes require less space and are faster regarding updates.

Primary vs Secondary Index

Definition (Primary Index). Indexed on primary key of the relation. An index whose search key specifies the sequential order of the file

Definition (Secondary Index). Indexed on candidate key of the relation.

Unique vs Non-Unique Index

Definition (Unique Index). An index over a candidate key is called an unique index. No duplicates are allowed.

Definition (Non-Unique Index). Duplicates are allowed for the index.

Remarks

- Every unclustered index must be dense.
- Every clustered indexd **need not** be dense.

Week 4 - Hash-Based Indexing and Bitmap Indexes

Hash-Based Indexes

We look at further techniques on how we support querying the data.

Hash-based indexes are better than tree indexes for equality selections. **Hash-based indexes does NOT support range searches**.

Table indexes are auxillary data structures that makes it easier for to find specific tuples.

Definition (Bucket). A bucket is a unit of storage containing one or more records that is stored in a page.

Index entries are partitioned into buckets based on a hash function h(v) = a where v is the search key value and a is the address to store the data.

Lemma (Equality Search with Hash Index). There are 3 steps to do an equality search with hash index. Given a search value v,

1. Compute h(v)
2. Fetch bucket at h(v)
3. Search the bucket for tuple of interest

Definition (Skewed data distribution). The distribution of hash values of data entries is not uniformly distributed.

Review of Hash Tables

A hash table is an associative array that maps keys to values. There are 2 parts to a hash table:

- **Hash Function**: Mapping a large key space into a smaller domain. There is a trade-off between fast execution vs collision rate for choosing what kind of hash function we want to use.
- **Hashing Scheme**: Handling key collisions after hashing. This is inevitable as we do not have a perfect hashing function. There is a trade-off between allocating a large hash table to reduce collisions vs executing additional instructions to find/insert keys.

Static Hashing

Open Addressing Hashing

The most naive approach. This is a single giant table of slots. We resolve collisions by linearly searching for next free slot in the table. The assumption we have made is that we know the number of unique keys to store and that our data is of a fixed length. Furthermore, we have assumed a **perfect hash function** which is impossible and expensive to maintain.

We try to approximate a perfect hash.

Linear Probe Hashing

Single giant table of slots. We remove collisions by linearly searching for the next free slot in the table. To determine whether an element is present, hash to a location in the index and scan for it. Insertion is cheap but update and deletion is heavily.

However, the issue is that we assumed non-unique keys. Therefore we move on to the next method that can deal with non-unique keys.

Static Hashing

Maintain a linked list of buckets for each slot in the hash table. Resolve collisions by placing elements with same hash key into the same bucket. If the bucket is full, just add a new bucket. Let `M` be the number of buckets. The hash-value is then `h(v) mod M` to find out which bucket does entry go into. Note that the number of primary pages is fixed.

An issue is that long *overflow chains* can develop and degrade performance. Furthermore, this technique is not space efficient. Note that this is something not common in industry.

Dynamic techniques can help solve this problem.

Dynamic Hashing

The general idea is to maintain a linked list of **buckets** for each slot in the hash table. We resolve collions by placing all elements with the same hash key into the same bucket. To determine whether an element is present, hash to its bucket and linearly scan for it.

For bucket overflows, we don't want to re-organize the file by just naively doubling the number of buckets as this will be a very expensive operation. Instead, we want to fix the bucket that has just overflowed.

Chained Hashing

We have a hash table and a linked list of buckets for each entry. If a bucket is full, we just add a new bucket to the end of the linked list. The issue is that these buckets can grow infinitely because we can just keep adding new buckets to the linked list. **Dynamic hashing** will help us solve this issue of lists of buckets from getting too long.

Extendible Hashing

Similar to chained hashing but instead of letting the linked list of buckets grow indefinitely, we split them up incrementally. Note that when hashing, we look at the **last** few bits to determine which bucket to get mapped to.

When a bucket is full, we split the bucket and reshuffle its elements. We use the global and local depth to help us determine the buckets to split up.

Definition (Global Depth). **Maximum** number of bits to examine across **all** slots in hash table that is needed to tell which bucket an entry belongs to.

Definition (Local Depth). Number of bits used to determine who from the hash table is pointing to you by looking at the local depth number of bits of things in this bucket. The pointers from the hashtable to the buckets are generated by the local depth where we look back at the hash table by the local depth number of bits and then we can figure out which entries in the hash table is pointing to us.

The local data is meta data for us to keep track of where we came from. Which entry in the hash table is pointing

to this bucket. The global depth is for the look up whilst the local depth is used to keep track of how we got there.

If the bucket is full, we increment the local depth, use the local depth to determine how we split up the entries and rehash elements in that bucket to new buckets. Note that since we are using bits, adding a new bit to look at for the local depth only gives us 2 possible buckets for the entry to go in. We are only rebuilding the bucket that has been overflowed, the rest of the buckets remain the same.

Furthermore, if the global depth is now less than the local depth, we double the hash table to allow for more buckets and increment the global depth.

The local depth is just for extending and not used to find what we are looking for. Once we are inside the bucket, we just sequentially look for the entry. When we split a bucket, the local depth will tell us how to split and whether to stay in the same bucket or into the new bucket.

Remark An alternative hashing schema in order to avoid the use of a large directory space is linear hashing.

Linear Hashing

Here we maintain a pointer that tracks the next bucket to split. When any bucket overflow, we **split the bucket at the pointer location**. Meanwhile, the bucket that has overflowed will just have an additional bucket attached to it and doesn't get split, just like in chained hashing. In extendible hashing, we split the bucket we just overflowed.

Definition (Split Pointer). A pointer to a bucket that we will split for when we overflow.

Steps for overflow:

1. Add **one** new directory in hash function.
2. Create new hash function based on n. We n = 4 and #buckets < 5, then our new hash function is `key % n`. But when n=5, we now change the hash function to `key % 2n`. When n=6, then `key %2n` is still the same. When we have 10 buckets, we now have `key % 3n`.
3. Then we split the bucket that our **split pointer** is pointing to by our new hash function and increment our split pointer.
4. Then we look at our original hash function and if that maps us to an entry above split pointer, then we use the second hash function to determine where to go.

- If hash function maps to slot that has previously been pointed to by pointer, apply the new hash function.
- When pointer reaches last slot, delete original hash function and replace it with new hash function.

Remark Linear hashing avoids directory by splitting buckets one at a time and using overflow pages. We can handle duplicates easily and it has better space utilization compared to extendible hashing.

Bitmap Indexes

The issue with hashing is that when we have a few distinct values, collisions will arise frequently.

Definition (Bitmap Index). A bitmap index for a field f is a collection of n-bit vectors where n is the number of possible values of f. The ith coordinate of the vector will take on the value 1 if the ith record has the value v in the field, and 0 otherwise.

Example Suppose we had a gender index with 3 possible values: Male,Female,Undetermined. The bit-vector representation for a female entry will be *010* whilst for a male it will be *100*.

Lemma (Space Complexity). The space complexity of bitmap index is `O(mn)` where n is the number of records and m is the number of distinct values for field f.

The issue with bitmap is that the more distinct values there are, the *sparser* the bitmap vector. Hence, we can compress the bitmap vectors.

Choosing Indexes

We have a trade-off where indexes can make queries go faster but updates slower and requiring space.

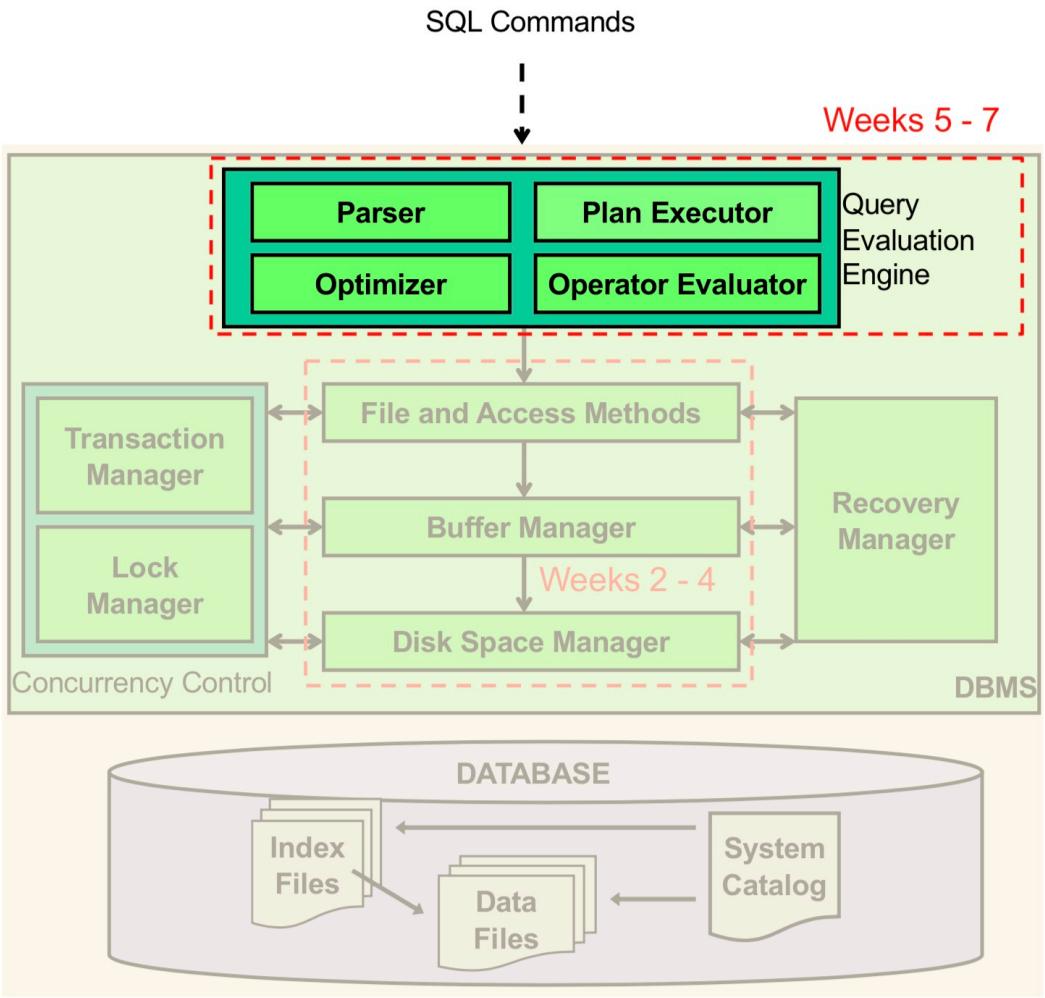
Heuristics for Index Selection

1. Choose attributes in `WHERE` clause.
2. Exact match conditions in `WHERE` clause suggests hash indices.
3. Range query conditions suggests tree index.
4. For `WHERE` clauses with several conditions, we can use a multi-attribute search key.

Week 5 - Query Processing and External Sorting

Role and structure of Query Processing

We are now looking at the query evaluation engine of the DBMS



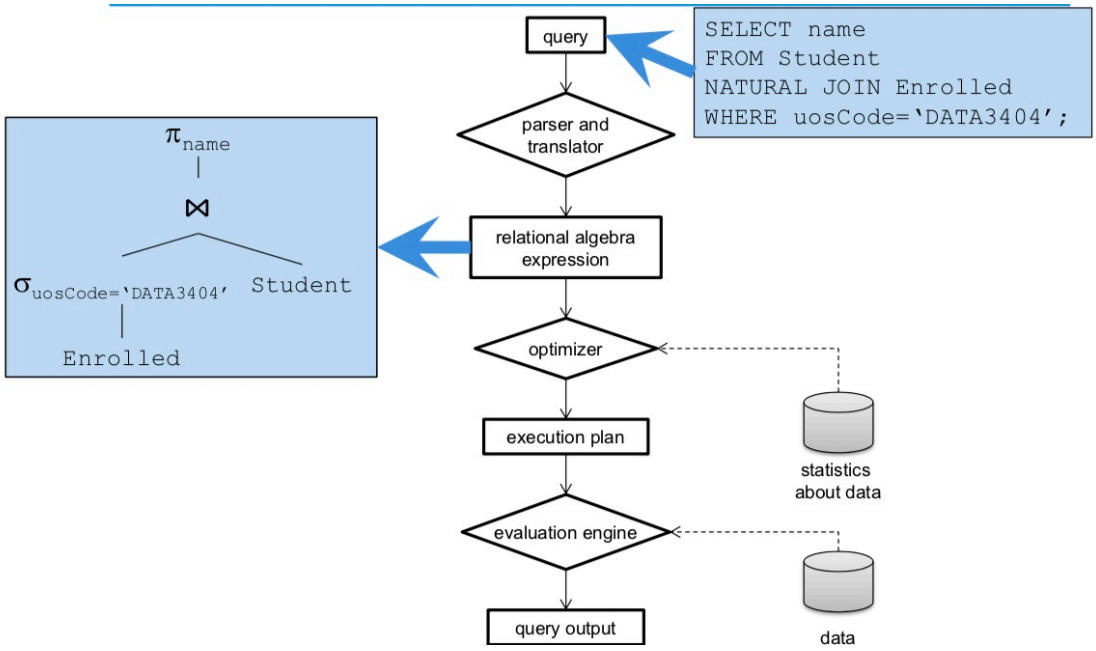
We have 3 steps from SQL to physical data access:

1. Query Parsing - Translating to physical access.
2. Query Optimization - Picking optimal plan to access data.
3. Plan Execution/Processing Models

1) Query Parsing

Lemma The **expression tree** of a query are just the semantics of the query whilst the **evaluation plan** of the query refers to the actual execution of the query.

The steps for query processing are as follows:



The DBMS converts a SQL statement into a query plan. The operators are arranged into a tree with data flowing from the leaves towards the root. The output of the root node in the tree is the result of the query.

Definition (Projection). A projection is a unary operator with attribute names $a_1, \ldots, a_n$ and relation R is expressed as

$$\pi_{a_1,\ldots,a_n}(R).$$

The result of the projection is the set of components of the tuple R restricted to the set $\{a_1, \ldots, a_n\}$.

Definition (Selection). A selection is an unary operator that denotes the subset of a relation. We define it as

$$\sigma_{a\theta b}(R)$$

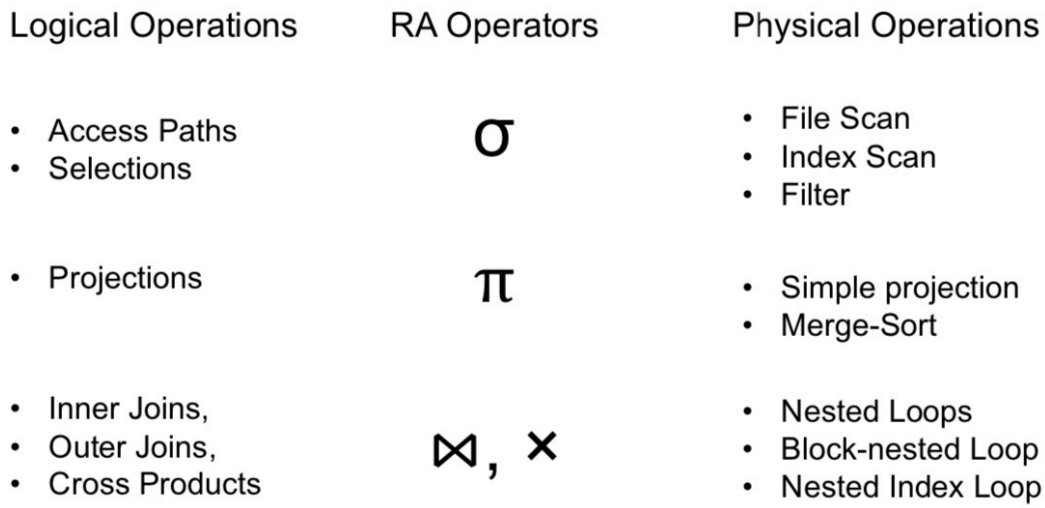
or

$$\sigma_{a\theta v}(R)$$

where

- a and b are attribute names
- θ is a binary operator in the set $\{<. \leq, =, \neq, \geq, >\}$
- v is a value constant
- R is a relation

We can see mappings from logical operations to relational algebra operators to physical operations



2) Query Optimization

There are two parts to optimizing a query.

a) Consider a set of alternative query plans

We need to prune the search space of query plans.

b) Estimate the cost of each query plan that is in our set

We need to estimate the size of result and the cost for each plan node. We uses statistics to help us with this.

We will look more into this in the next chapter.

3) Plan Execution

We now need to look at how to pass records between operations as we move up the expression tree.

Definition (Processing Model). A DBMS processing model defines how the system executes a query plan. There are different trade-offs for different workloads. We can use a top down or bottom up approach.

We look at 2 processing models.

Pipeline/Iterator Model (Tuple-at-a-time)

This is the most common implementation in DBMS, including Hadoop. It is always applicable but has an expensive I/O. Each query plan operator implements a `next` function.

1. Evaluate one row of output of a relation.
2. **Pipeline** each row to next operation.

This is an example of top down processing. The root node calls next on the tuples on its children nodes, but then those children nodes may need to call `next` to get the tuples for its children. We then need to traverse tree to very bottom.

The advantage is that it is much cheaper memory wise and we can process a tuple through as many operators as possible before having to retrieve the next tuple. The issue is that some operations which requires all tuples won't

work such as `SORT` and `JOIN`. They are known as *pipeline blocking* operations. This is also easy to control the output and limit the number of outputs.

Materialization Model (Set-at-a-time)

Each operator processes its input all at once and then emit its output all at once. VoltDB uses this approach. This is a bottom up approach. We `materialize` the output as a single result. Each operator does all the work it needs to do and then outputs a result. We never go back to a lower operator again as it should have done all the work it needs to do.

1. Evaluate whole operation.
2. Store (materialize) result into temporary relation.
3. Next operation reads in temporary relation.

This is better for OLTP workload because queries only access a small number of tuples at a time and the output from each operator will be small. This is not good for OLAP as you will end up with large intermediate results and move more data than necessary.

Remark There is a *vectorization model* which is similar to the iterator model but we emit a *batch (vector)* of data instead of a single tuple to be processed. Hence it is a top down approach too. This is good for OLAP queries.

Access Methods

Definition (Access Method). This tells us the way that the DBMS can physically *access* the data stored in a table. This is the bottom part of our query plan. This will be the bottom operators in a query plan that feeds data into the operators above it in the tree.

We have 3 approaches:

1. Sequential Table Scan
2. Index Scan
3. Multi-Index/Bitmap Scan

1) Sequential Table Scan

For each page in table, retrieve it from the buffer pool and iterate over each tuple and check whether to include it. The DBMS maintains an internal to keep track of the last page examined.

2) Index Scan

The DBMS picks an index to find the tuples that the query needs. We can use `WHERE` condition to retrieve a smaller set of tuples. This helps us to not require using sequential scan to access the data. So we note the predicate in `WHERE` clause and note what attributes in the index and then use your hashmap/tree to get the data.

We want to pick indexes that filter out as much data as possible to help save us time.

Query Execution Algorithms

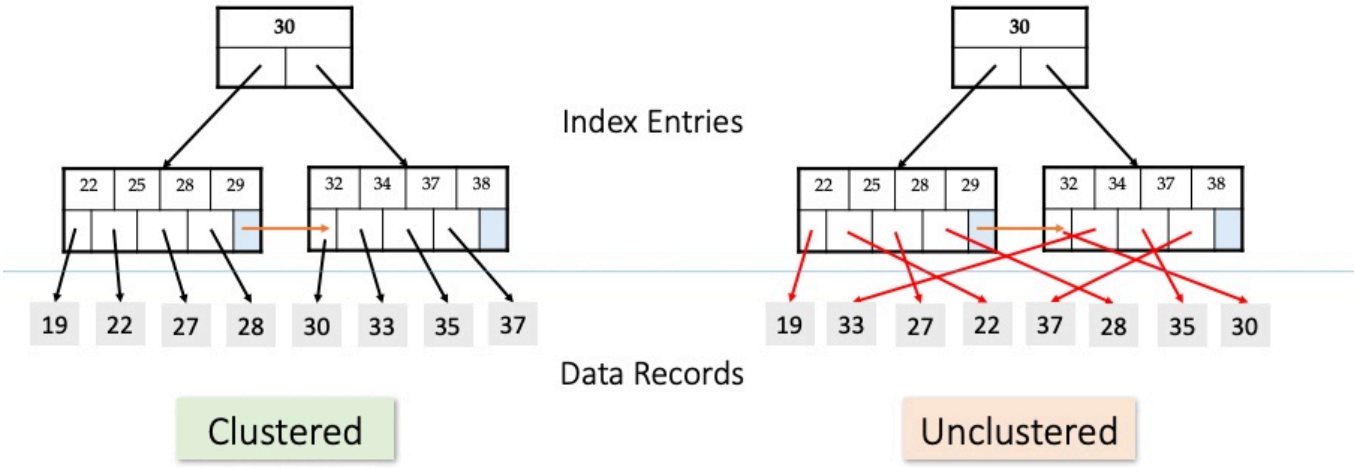
External Sorting Algorithms

We need sorting because in the relation model, tuples do have a specific order. Sorting is generally required for `ORDER BY`, `GROUP BY`, `JOIN`, and `DISTINCT` operators.

Sorting in DBMS has its own issues as the memory to store the data is not big enough. Hence, we need novel techniques to deal with this.

If the data fits in memory, we can run an algorithm like `quicksort` to help us. If it does not, we need to use *external sorting* that allows us to read and write to disk as needed.

We can accelerate sorting using a clustered B+tree by scanning the leaf nodes from left to right. This is a bad idea, however, if we use an unclustered B+tree to sort because it causes a lot of I/O reads (random access through pointer chasing). Recall that a clustered index is when the file is sorted on the index attribute. Furthermore, for a clustered B+ tree index, the data entries are the data records. So if we had a B+ tree with index on the key we are interested in, we can use the B+ tree instead of using external merge sort.



External Merge Sort

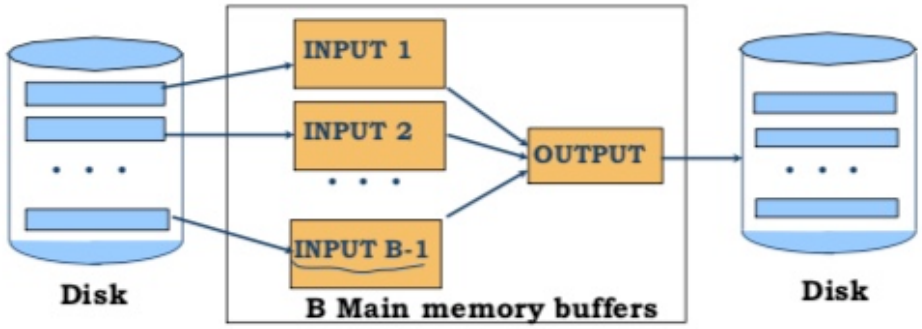
The task is to sort $N \in \mathbb{N}$ pages using $B \in \mathbb{N}$ buffer pages. The ideas is to load the pages into the buffer and sort `B-1` amount of pages at a time with the last buffer frame being used for the output of the solution. We then write the output out to disk. Then we merge our results.

Phase 1: Sorting - Sort small chunks of data that fit in main memory and then write back to disk.

Phase 2: Merge - Combine sorted subfiles into a larger single file.

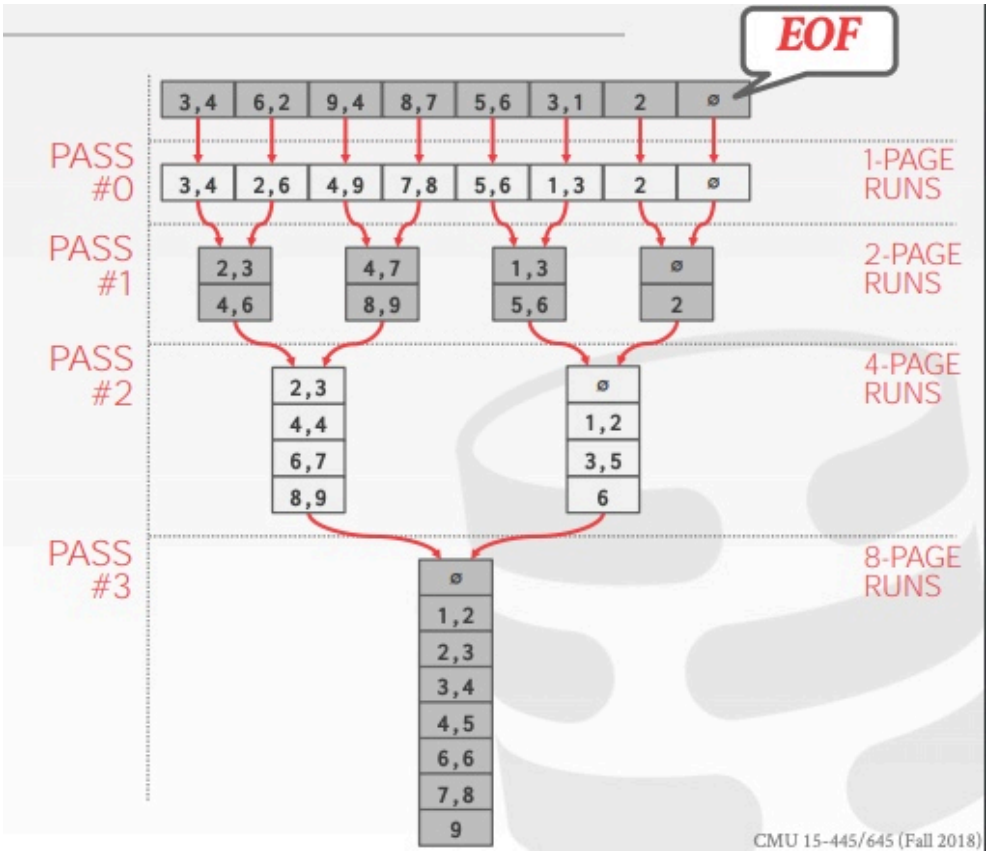
Definition (Run). Each sorted set of pages is called a run.

We have to run this multiple times until we reduce down into a final sorted solution. For a given run, it looks like this



Pseudocode

1. Use B buffer pages. Produce N/B sorted runs of size B pages each.
2. Recursively merge the B-1 runs and write back to disk.
3. Repeat until we have sorted pages.



Lemma The number of passes made is $\lceil Log_{B-1}(\frac{N}{B})\rceil + 1$.

Lemma (I/O Complexity). The I/O complexity for this is

$$2N\Big(\lceil Log_{B-1}(\frac{N}{B})\rceil + 1\Big) = 2N \circ (\#\text{ of passes})$$

where N is the number of pages to sort on.

We need to read in and read out.

Larger block size means less number of runs require to merge.

Week 6 - Query Evaluation (Join Algorithms)

Query Execution Algorithms

Revision of Joins

For good database design, we want to minimize the amount of information repetition. This is why we compose tables based on normalization theory. Joins are required to help us reconstruct original tables.

Definition (Join). For a tuple $r \in R$ and a tuple $s \in S$ that match on join attributes, the join operator $\bowtie$ concatenates r and s together into a new output tuple.

Definition (Theta Join). Rows from R and S combine if some condition θ is true. $R \bowtie_{age>minimum} S$.

Definition (Equi-Join). Rows from R and S combine if specified attribute match in value.
$R \bowtie_{firstName=nickName} S$.

Definition (Natural-Join). Rows from R and S combine all attributes of **same name** match in value. $R \bowtie S$.

Remark A natural-join is a special case of equi-join.

Definition (Inner Join). Rows from R and S combine on attributes of column names mentioned in the `ON` clause.

Definition (Outer join). An inner join combined with additional results that are not part of the inner join. We have 3 types of outer joins: left outer join, right outer join, and full outer join.

Definition (Full outer join). Return all records from both tables and matching records if there are any matches.

Definition (Left outer join). Return all records from the left table connected to any matching records in the right table.

Definition (Right outer join). Return all records from the right table connected to any matching records in the left table.

Remark The cross product operator $R \times S$ followed by a selection is inefficient because the cross-product is large and expensive. There are no particular algorithms that can be used to reduce the cost of cross product.

Example *SQL queries using joins*

```sql
SELECT * FROM TableOne INNER JOIN TableTwo on TableOne.Name = TableTwo.Name;

SELECT * FROM TableOne LEFT JOIN TableTwo on TableOne.Name = TableTwo.Name;

SELECT * FROM TableOne RIGHT JOIN TableTwo on TableOne.Name = TableTwo.Name;

```

```sql
SELECT * FROM TableOne FULL JOIN TableTwo on TableOne.Name = TableTwo.Name;
```

Remark *A **natural join** tries to identify the join criteria to be where the columns have the same name.*

Remark *An alternative to the **ON** keyword is the **USING** keyword. This is useful for when we are joining and the column names we want to join on are the same.*

Physical Join Implementations

There are many different physical implementations of the join algorithms.
-Nested-loop join
-Block nested loop join
-Index nested loop join
-Sort-Merge join

We measure their costs by counting the number of I/Os incurred by reading data from disk. We don't know what the output because that actually depends on the actual data itself, hence so we can't measure output costs.

We will be working off table `table_R` and `table_S` through the next cases.

Nested-Loop join

These are the most general form of joins. Whilst the most inefficient, it works for any join condition.

```
result_tuple = []
for tuple_R in table_R:
  for tuple_S in table_S:
    if join_condition(tuple_R, tuple_S) == True:
      result_tuple.append(tuple_S) # can be right or left tuple
```

Remark Setting the right_relation to be the smaller relation will save time.

Lemma (I/O Cost). Let `R` be the number of records in R, b_R size of the table R and b_S be the size of the table S. Then the cost is $|R| * b_S + b_R$.

Remark Pretty naive because we can pack multiple tuples in a single page but here for every tuple, we do a page fetch.

Remark For each tuple in the outer table, we do a sequential scan to check for a match in the inner table.

Block Nested-Loop Join

Every block (sequence of pages) of the inner relation is paired with every block of the outer relation. For each block in the outer table, fetch each block from the inner table and compare all the tuples in those two blocks. This algorithm performs fewer disk access because we scan the inner table for every outer table block instead of for every tuple.

```
result_tuple = []
for block_R in table_R:
```

```
  for block_S in table_S:
    for tuple_R in block_R:
      for tuple_S in block_S:
        if join_condition(tuple_R, tuple_S) == True:
          result_tuple.append(tuple_S)
```

Lemma (I/O Cost). Let b_R size of the table R and b_S be the size of the table S. Let M be a buffer manager of M pages. Then the cost is $|b_R| * b_S + b_R$.

This only uses 2 buffer frames. We can instead use B-2 buffers to scan in `table_R`.

Lemma (I/O Cost). Let b_R size of the table R and b_S be the size of the table S. Let M be a buffer manager of M pages. Then the cost is $|b_R/(M-2)| * b_S + b_R$.

Ultimatley, both block nested-loop and nested-loop joins are just sequential scans, which is inefficient. Why not use indexes?

Index Nested-Loop Join

We can use an index to find *inner table matches*. We can use existing indexes for the join or build one on the fly (hashmap). We can just probe the index instead of sequentially scanning the inner table.

This join has 2 conditions:

1. We are doing an equi-join/natural join
2. An index is available on our inner relation's join attribute.

Now, instead of doing a file scan of the inner relation, we can use the index to look up tuples in the inner relation that satisfies the join condition with the outer relation.

```
result_tuple = []
for tuple_R in table_R:
  for tuple_L in Index(tuple_L = tuple_R): # use index to find matching tuple
    result_tuple.append(tuple_L)
```

Assume cost of each index probe is a constant C per tuple.

Lemma (I/O Cost). Let b_R size of the table R and b_S be the size of the table S. Then the cost is $|R| * C + b_R$.

Remark Pick the smaller table as the outer table and buffer as much of the outer table in memory as possible.

Sort-Merge Join

This join has 1 condition:

1. We are doing an equi-join/natural join.

There are 2 steps to this join:

1. Sort the two tables on the join attribute. We can use the external merge sort algorithm or any other sort algorithm.
2. Scan the two sorted tables in parallel and match tuples.

```
result_tuple = []
sort(R, join_key) # outer table
sort(S, join_key) # inner table

cursor_r = R_sorted
cursor_s = S_sorted

while cursor_r and cursor_s:
  if cursor_r > cursor_s:
    cursor_s += 1
  elif cursor_r < cursor_s:
    cursor_r += 1
  else:
    # They match
    result_tuple.append(cursor_s)
    cursor_s +=1
```

So we increment the side that is smaller and if they match, then we increment the inner table cursor. Once we reach the end of one table, we are done.

Lemma (IO Cost). Let b_R size of the table R and b_S be the size of the table S (number of pages). Cost is: $b_r + b_s$

Hash Join

This join has 1 condition:

1. We are doing an equi-join/natural join.

If two values are the same, then their hashed values should be the same.

There are 2 steps to this join:

1. **Build** - Scan the outer relation and populate a hash table using the hash function on the join attribute.
2. **Probe** - Scan the inner relation and use the hash function to jump to find matching tuple.

We can use a **static hash table** if we know the size of the outer table or else we need to use either a **dynamic hash table** or allow for **overflow pages**.

The hash join is the fastest algorithm.

```
result_tuple = []
hash_table_R = build_hash_table(R)
for tuple_s in table_S:
  if hash(tuple_s) in hash_table_R:
    result_tuple.append(tuple_s)
```

The hash table has 2 options for values:

1. Full tuple inside the hash table value. Saves us time but more memory.
2. Tuple identifier inside the hash table. Saves space.

Grace Hash Join

This is for when tables do not fit onto main memory. This extends on hash join so that it also hashes the inner table into partitions that are written out to disk.

1. **Build** - Scan **both** outer and inner tables and populate a hash table using the hash function h_1 on the join attributes. We write the hash table buckets to disk as needed. If a single bucket does not fit in memory, then use *recursive partitioning* with a second hash function h_2 to further divide the bucket.
2. **Probe** - For each bucket level, retrieve the corresponding pages for both outer and inner tables. Then perform a nested loop join on the tuples in those two pages. The pages will fit in memory, so this join operation will be fast.

Lemma (IO Cost). Cost is $3 \times (b_r + b_s)$ since partitioning is $2 \times (b_r + b_s)$ and probe phase is $(b_r + b_s)$.

Aggregations

Definition (Aggregations). This is when we collapse multiple tuples into a single scalar value.

We have two implementation choices:

1. Sorting
2. Hashing

Hashing is generally the better approach unless there is an `ORDER BY` statement.

Sorting

Distinct

We can filter and then project to get attribute we want distinct values for. Then we can sort the attribute and iterate through it whilst we keep a buffer to keep track of what was the last value we iterated through.

Hashing

Generally hashing is used instead since we may not necessarily need the data to be sorted. Getting the data sorted can be costly.

Distinct

We just hash the value and see if value already exists in the hash table. This is simple if the data is in-memory.

Remark The hashing methods section assumed that we can hash data in memory. There are techniques for external hashing aggregate techniques.

Hashing Aggregate

Phase 1: Partitition Use a hash function h_1 to split tuples into partitions on disk based on target hash key. We then write out to disk.

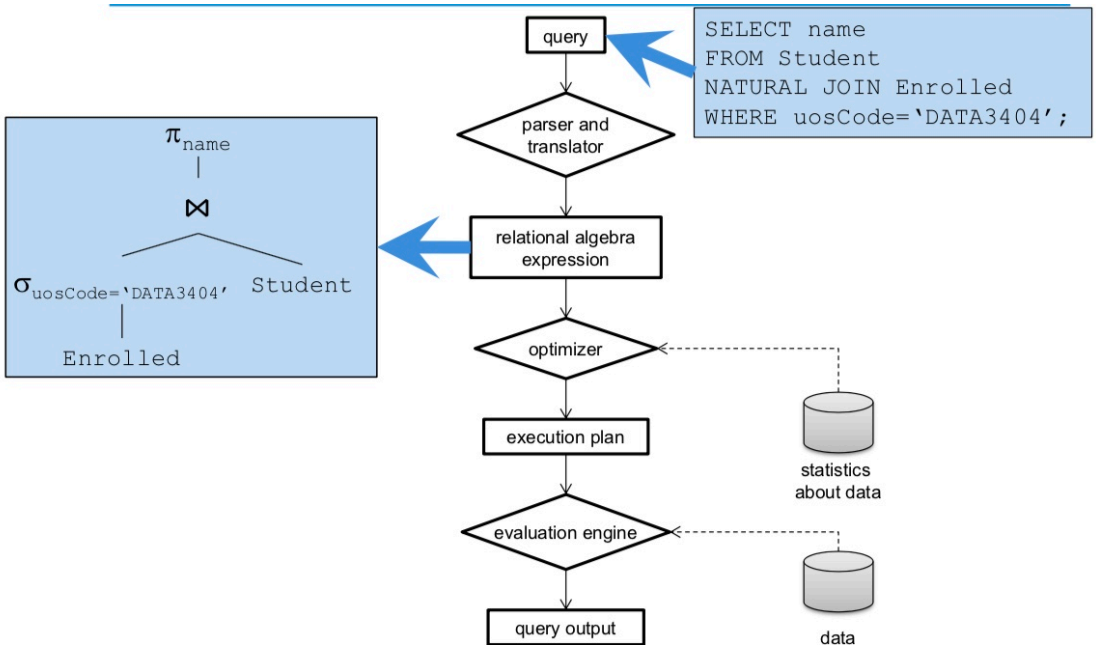
Phase 2: ReHash For each partition on disk, read pages into memory and build an in-memory hash table based on a second hash function h_2. Then go through each bucket of this hash table to bring together matching tuples and compute the aggregation.

Week 7 - Query Optimization

Query Optimization

Motivation

Recall the steps for query processing:



Recall that SQL is declarative. The DBMS needs to translate a SQL statement into an **executable query plan**. However, there are numerous **logically equivalent** executable query plans. Hence, we need to find the optimal one.

We write high level SQL queries and let the query optimiser to work on optimising the query.

Optimization Strategies

1. **Heuristics/Rules**: Humans rewrite query to remove inefficiences. We try to minimize the tuple and column counts of the intermediate and final query processes. We do not need a cost model as a result.
2. **Cost-based Search**: We can use a cost model to evaluate multiple equivalent plans and pick the one with the minimum cost.

The optimizer will never pick the **global** minimizer but attempt to prune the search space to find a decent local minimizer.

Query optimization is NP-hard.

Rule-based Query Optimization

Definition (Equivalence). Two relational algebra expressions are equivalent if they generate the same set of tuples.

Heuristics to use

We can categorise the heuristics of query rewriting as follows

- **Predicate Push-Down** Perform selection early before join to reduce the number of tuples.
- **Projections Push-Down** Perform projections early on to create smaller tuples and reduce intermediary results. We can project out all attributes that are not needed (like the join attributes for example). Reduce the number of attributes.
- **Expression Simplification** We can use the transitive properties of boolean logic to rewrite predicate expression into a more simple form.
- **Restrictive** Perform most restrictive selection and join operations before other similar operations.

More specifically, we can use the following properties of relational algebra to help us with query optimization.

Definition (Sigma-cascade). The conjunctive selection operation can be written as a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}\left(\sigma_{\theta_2}(E)\right)$$

Remark Intersection of selection operators is an expensive task. Apply the selection operator that yields less tuples first.

Definition (Commutative-selection). We can commute the selection operator.

$$\sigma_{\theta_1}\left(\sigma_{\theta_2}(E)\right) = \sigma_{\theta_2}\left(\sigma_{\theta_1}(E)\right)$$

Remark It is better to apply the selection condition that yields less tuples first.

Definition (π-cascade). We can reduce a series of projections down into the last projection operator to be performed.

$$\pi_{L_1}(\pi_{L_2}(\ldots(\pi_{L_N}(E))\ldots)) = \pi_{L_1}(E)$$

Remark The columns arising from the π_{L_1} projection are the only ones that will be viewed. Hence we should collapse all projections into the outermost projection.

Definition (Projections and selections). A projection commutes with a selection that only uses attributes retained by the projection. That is, we can commute projections and selections if the selection uses attributes required for the projection.

Definition(Cartesian products as theta-joins). We can express the selection on Cartesian products as θ-joins.

$$\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$$

Remark The cross product operation is extremely expensive and furthermore, the application of the selection operator on it will increase the runtime even further.

Definition (Join operators are associative and commutative). We can apply join operators that yield the less tuples first.

$$\left(E_1 \bowtie E_2\right) \bowtie E_3 = E_1 \bowtie \left(E_2 \bowtie E_3\right)$$

$$\left(E_1 \bowtie E_2\right) = \left(E_2 \bowtie E_1\right)$$

Definition (Selection operation is distributive). Join operators are expensive so joining smaller sets of tuples will be faster.

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie E_2) = \left(\sigma_{\theta_1}(E_1)\right) \bowtie \left(\sigma_{\theta_2}(E_2)\right)$$

Definition (Projection distributes over θ-join).

$$\pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \left(\pi_{L_1}(E_1)\right) \bowtie_\theta \left(\pi_{L_2}(E_2)\right)$$

Left-Deep Join PLans

Left-deep join trees can help us narrow down search space regarding joins. We can use pipeline plans regarding the data. In a left-deep join tree, the right hand side input for each node in query tree is just a relation, not the result of an intermediate join.

Cost-based Query Optimization

We can use an internal cost model to **estimate** the execution cost for a particular query plan. This means we can use these estimates instead of actually running all the possible query plans which will take a long time.

The steps to generate query-evaluation plans for an expression are:

1. Generate logically equivalent expressions
2. Annotate resultant expressions to get alternative query plans
3. Choose the cheapest plan based on estimated cost

2 main issues with cost based query optimization:

1. What plans are considered for a given query?
2. What is the cost the plan estimated?

Costs for query evaluation plans

The features that go into the metric are:

- **CPU**: Small cost.
- **Disk**: Number of blocks transferred.
- **Memory**: Amount of DRAM used.
- **Network**: Number of messages trasnferred

The DBMS stores internal statistics about tables, attributes, indices etc in its internal catalogue. For this course, we only look at number of page transfers from disk (I/Os) as the cost measure.

Assumption: We assume that the data values are uniformly distributed when estimating costs.

We need to estimate:

- Cost of each operation in the plan tree.
- Size of result for each operation in the tree.

To maintain database statistics about this, we utilise **periodic sampling**.

Week 8 - Distributed Data Management

Distributed Data Management

Motivation

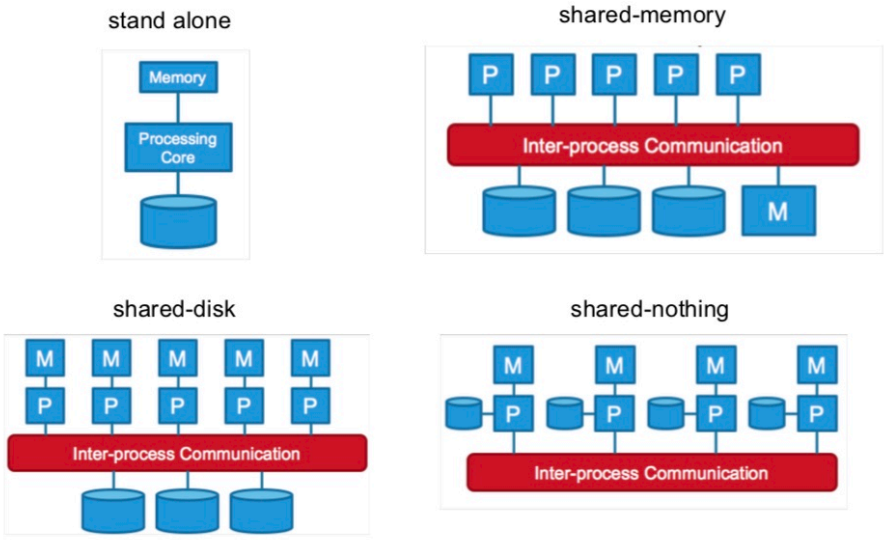
So far we only considered **stand-alone** server. For real big-data processing, we need to scale out to a cluster of multiple servers (nodes). So we now scale out to more machines but alongside it, we now need to consider issues faced in distributed systems.

Distributed vs parallel database

- A parallel DB are ones where machine are physically close to each other and communicate via high speed connection. The architecture can be of any form. Meanwhile, distributed DB are more generalised than parallel databases and not necessarily in close proximity. They are usually forced to use shared nothing architectures.

Architectures

Below are the four main architectures for a parallel system.



Shared memory: We all share the memory. Only a single address space for both disk and memory. CPU have access to common memory via an interconnection network. Each processor has a global view of all in-memory data structures. Each DBMS instance needs to know about other instances. *No one uses this as this architecture is not scalable beyond 64 processors due to the interconnection network becoming a bottleneck.*
Shared disk: You share the main storage but each node has its own memory. These memory are in fact stateless and does not store data. Modern setups can have caches in between disk and memory. They all access the same physical location. Storage shared between computational nodes. If we want to update/write to the main memory, we need to let other nodes know what we are doing. The nodes are stateless. If we want to scale up storage, we can just add more disk space and if we want to scale up processing, just add more nodes. This is the most popular architecture. It provides a good degree of fault-tolerance as if a processor fails, other processors can take over

tasks. However, the bottleneck is now at the interconnection to the disk subsystem.
Shared nothing: Each CPU has its own memory **and** disk. We now need to send messages between the systems to let them know what we're doing. Each node has a single cpu, memory, and disk. Now we need to go over the network to let other nodes know what's happening. It is easy to increase capacity but hard to ensure consistency. Now we also need to coordinate on the state of the disk which increases the cost of communication and also non-local disk access. Examples of this architecture include Redis, Greenplum and Cassandra.

Design Principles for architecture

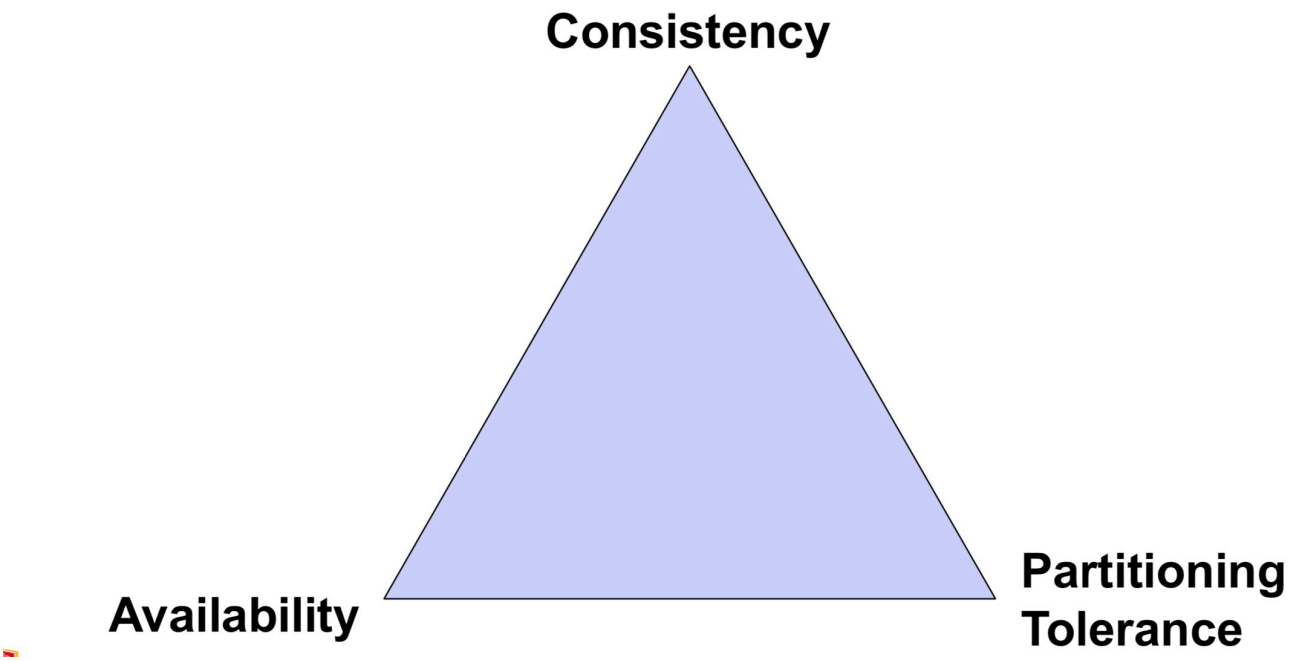
The goals for a shared data system are

1. **Consistency**: "All-or-nothing" semantics. Additionally, all copies have the same data.
2. **Availabililty**: The data system should always be up. However, the larger the system, the higher the probability of failure.
3. **Partition Tolerance**: If there is a network failure that splits the processing into two disjoint subsets, then the goal is to ensure the processing continues in these subgroups.

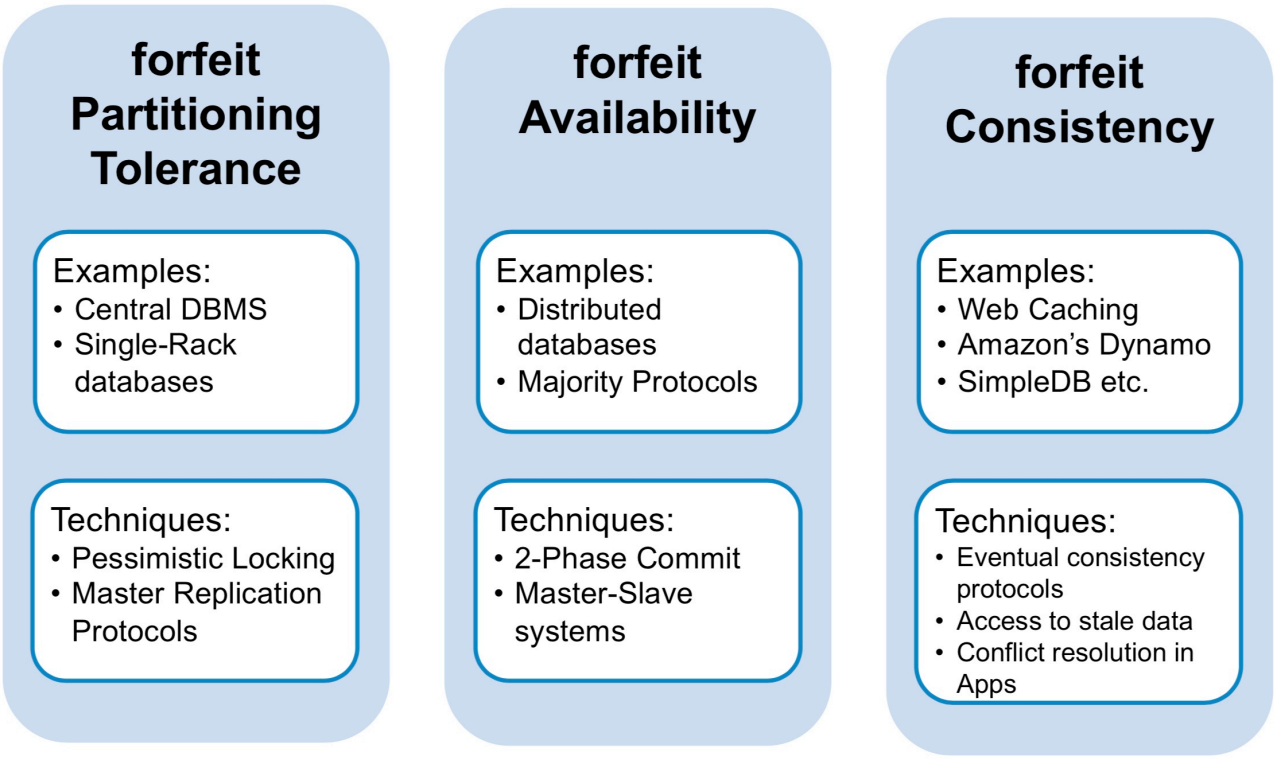
Network failure is extremely high and alot more common than you think.

WAN latency is alot higher compared to LAN network.

Theorem (Cap Theorem). You can have at most **two** of the properties for any networked shared data system.



If you want your system to be partitioning tolerance, then if your system has a failure, you can either let you system to still be available to use but run the risk of not being consistent and vice versa.



Note that cloud serivces tend to want to satisfy availability and partition tolerance as that is more of a hardware problem.

Physical Design Techniques

There are **two** main physical design techniques. Data partitioning and replication.

(A) Data Partitioning

Identify subsets of data and store them at different places. The aim is to get smaller data sets and hence it is easier to scale by parallelism.

You can partition by columns (vertical partition) or by rows (horizontal partition). If each partition is stored on a different site, this is known as **sharding**. Note that sharding applies for horizontal partitioning (rows) where a row is known as a **shard**.

There are 3 ways to **place the partitioned data into nodes**, that is, how do we distribute the partitions across nodes.

1. *Round robin* - Multiple nodes available and place the partitions into nodes on a round robin basis. With n partitions, the ith tuple in insertion order is assigned to partition (i mod n). It ensures uniform data distribution. However, direct access to individual tuples to be tricky.

2. *Hash partition* - We have a hash function to which we apply onto the tuple id/key. Here, we apply the hash on the **data itself** to determine where should it go. It is useful for matching queries.

3. *Range partition* - Each node stores a partition defined by a **range predicate**. We use the data itself to tell us how to store the data. It is useful for both range and matching queries. However, skewed data will lead to terrible load balancing when distributing the data based on a range.

When we refer to data partitioning, this can refer to partitioning data on the same or across multiple machines whilst when we use the term data sharding, it always refers to multiple machines. In MongoDB, we partition the

data across different nodes.

Advantages of partitioning against data sharding:

- Better availability. If one partition is down, others are unaffected if stored on different disk.
- Easier to manage than a large table
- Helps with bulk loading.
- Queries faster on smaller partitions and can be evaluated in parallel.

Definition (Fragmenting/Data Sharing). Distributing data partitions over several sites in a distributed system. Here, we assume that queries access only one shard at a time.

Since we now have data on different places, we have two types of queries as a result of data partitioning.

1) Inter-Query Parallelism

We have multiple queries on different cores/processes.

2) Intra-Query Parallelism

We take the operations of a single query and parallelize this. We can do it either by

- **Horizonal Parallelism**: We apply the same operator in a query across threads. The query plan's operators are decomposed into independent instances that perform the same function on different subsets of data. The DBMS inserts an exchange operator into the query plan to coalesce results from children operators. The exchange operator prevents the DBMS from executing operators above it in the plan until it receives all of the data from the children.
- **Vertical Parallelism**: We take 2 different operators in the same query onto 2 different threads. Operations are overlapped in order to pipeline data from one stage to the next without materialization. This is sometimes called pipelined parallelism. Not all operators can emit output until they have seen all of the tuples from their children. This is more common in stream processing systems, systems that continually execute a query over a stream of input tuples.

Distributed Query Optimization

When we look at query optimisation for a distributed setting, here, we consider all the possible plans and pick the cheapest one. There are a few differences now compared to the non-distributed setting as

1. Communication costs must be considered.
2. Local site autonomy must be respected.
3. New distributed join methods.

The query site constructs a global plan with suggested local plans. That is, we figure out which places we need to go and how are the actual joins performed at those locations.

Distributed Query Evaluation

There are 2 query evaluation strategies for a distributed setting.

1. Materialization: For each operator, materialize the complete intermediate result. Then send the the result to the target node. This is good for a distributed setting as we can create temporary results to pass onto another node.
2. Pipelining: Items produced one at a time with no temporary files. We introduce meta-operators into the query plan which hides the communication. This is known as the **exchange** operator. This is used in

apache Flink.

Distributed Join

How to join tables if they are distributed among a cluster of nodes?

1) Local (Replicated) reference tables

If one of the join table is small and stable, replicate it onto all cluster nodes. Then take the big table and partition it across all the nodes and do local joins on each of the local nodes. Then do a local join on each node.

2) Broadcast join

If one table has a filter condition, then broadcast filtered tuples to all nodes and do subsequent local joins per node. Here we do it on the fly by filtering and broadcasting the data.

3) Distributed-shuffle join

This requires an equi-join or natural join. The modern technique is a distributed hash join. We shuffle/sort input data and then re-distribute to 1 node based on partitioning.

4) Fragment-and-Replicate join

Fragment both input relations. Replicate fragments to multiple join nodes. This works for any join condition.

(B) Data Replication

Assuming you have failures, we want to **optimise for availability** of the data. Hence, we will be storing copies/replicas of the same data at more than one place. The goal here is fail safety/availability. If one site is down, another replica can still be read. A beneficial side effect is that load balancing (parallelism) for fast read-only queries is now improved too. We can look up data across different nodes in parallel now. As a result, we have better load balancing for faster read only queries.

Updating Distributed Data

Updating distributed data depends on whether did we partition (split up) or replicate our data across many different sites.

1. Data partitioning: Just go to the site with the data and update it. If more than one site, an atomic commit protocol is needed.
2. Data replication: We now have seral design choices on how to update data that has been replicated..
   2a) **Synchronous Replication**: All copies of modified relation must be updated before modifying the transaction commits. We update all copies at the same time.
   2b) **Asynchronous Replication**: Copies of a modified relation are only periodically updated. Different copies may get out of sync. This violates consistency.

Key principle for replication: You should be able to read any copy. This links to availability. However, when we change data, we have issues. Should we edit all the data? If so, we can run into fault-tolerance and performance. Hence, data replication is best for data when we mainly want to use read operations compared to updates.

Where to replicate data:

- Total replication: All dbs have identical content. Simple to design but bad performance since we need to worry about updates. However we can now read anywhere.

- Partial replication: We only replicate certain parts of dbs. We need to manage information about replica locations. Need to make choices regarding data placement. This is more complicated but performance can now improve. We can replicate copy of some columns.

For our partial replication, what should we replicate?

- Complete tables: Each We replicate all the information.
- Fragments of tables: We only replicate subset of data.

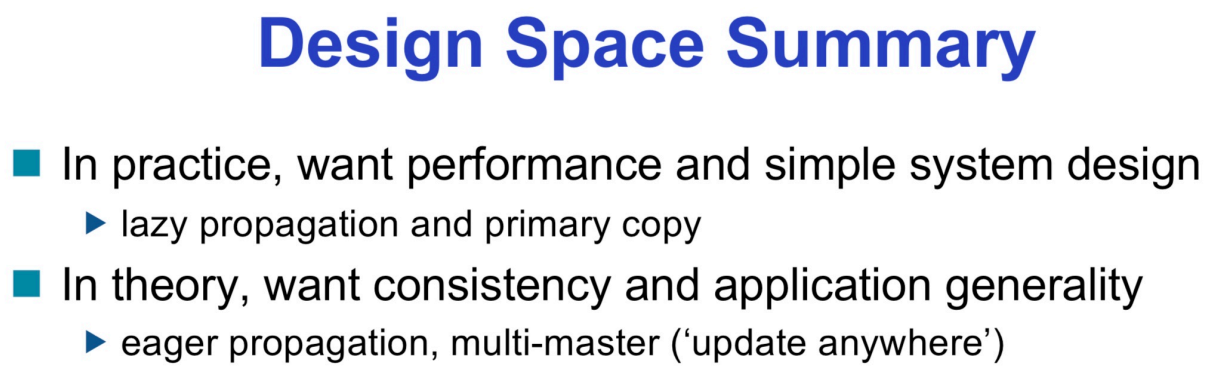
Replication Consistency

- Always consistent: Transparent replication where applications should not observe difference from single DBMS.
- Eventually consistent: Convergent. Eventually all copies will reach common values.

When to propage writes to replicas?

- Eager replication: Synchronous replication. So update during the transaction itself. Update replicas inside original transaction. Good for consistency but bad for performance.
- Lazy replication: Asynchronous replication. Update only one copy. Then propage changes later on. Good for performance but bad for consistency.

Primary Copy vs Multi-Master for Lazy Replication
Primary copy: Here we have a master-slave architecture. So we have one copy designated as the master. The other sites subscribe to the relation and are referred to as secondary copies. We update this master and all other nodes will copy/replicate it. It is bad for flexibility but good for order and consistency.
Multi-Master: We can update any node. Different transactions can update replicas in different orders. Good for flexibility. More than one of these copies of an object can be a master and any changes made must be propagated to other copies. However, inconsistencies can now arise.



	Primary Copy	Update Anywhere
Lazy Propagation (Asynchronous)	most practice approaches	
Eager Propagation (Synchronous)		ideal world

Ideally we want to eager propagation and being able to update anywhere but this does not work well in the real

world.

So how can we go about doing synchronous replication?

- Voting: Transactions must write a majority of copies to modify an object and read enough copies.
- Read-any Write all: Writes are slower and reads are faster relative to voting.

However the costs of synchronous replication are deadlocks and issues with group communication protocols.

Distributed Queries and replication

HDFS (Hadoop File System)

This is a new type of distributed file system. Stable storage. We have higher-level frameworks such as MapReduce/Hadoop on top of HDFS. Furthermore, there is spark and flink. NoSQL is also another alternative to storage.

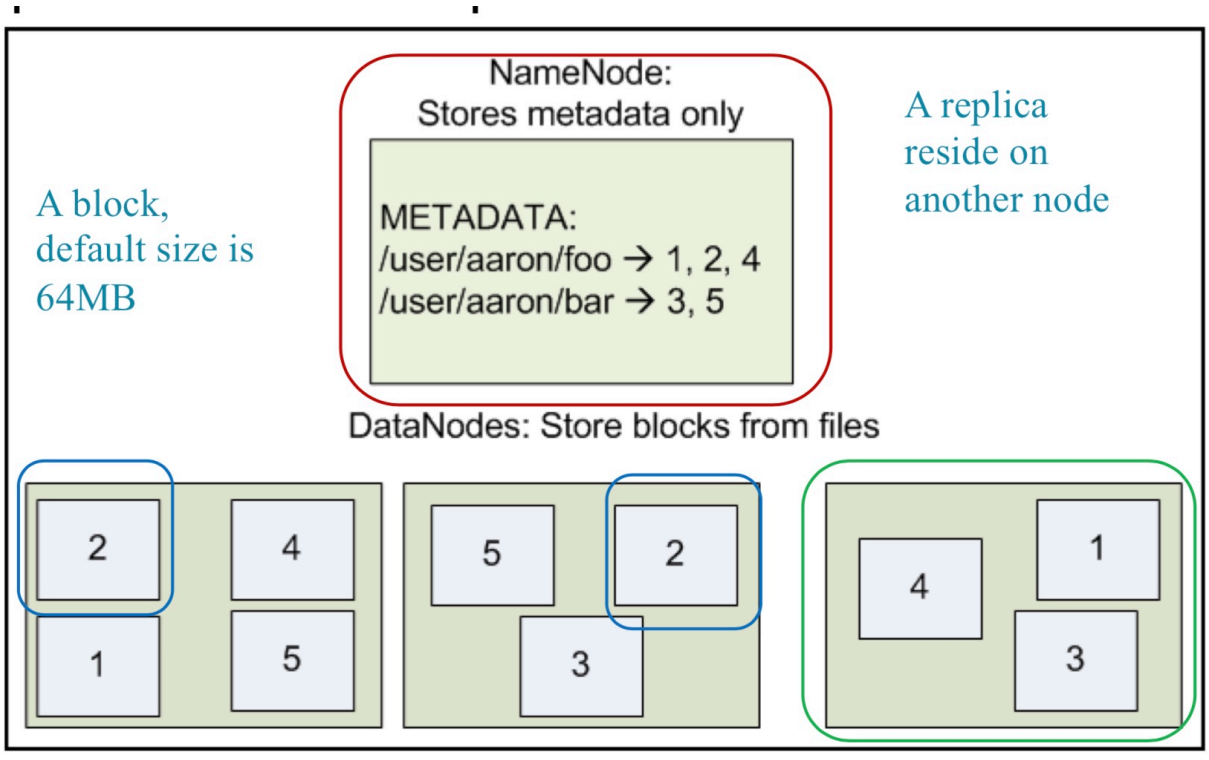
A distributed file system allows clients to access files on remote servers **transparently**. A limitation is that there is no sufficient mechanism in terms of reliability and availability.

Requirements for big data analysis

- Scalability - Support for both small and huge datasets.
- Reliability
- Consistency is not important. Data is usually written once but read multiple times.

HDFS

- Cluster with master-slave architecture. Dedicate master node to dictate where the data is etc. The slave nodes is the actual data storage and reads/write data based on the master slave. There is a **TCP** communication going on between clients and nodes, typically from master to slave.
- We usually partition HDFS files into large blocks (64 MB). Each block replicated across multiple nodes.



The NameNode (master node) has 3 types of metadata:

1. File/Data block namespaces
2. Mapping from files to data and back
3. Locations of data block replicas

All metadata is in memory. There is a large block size (64mb) to ensure small meta data size. First two are kept persistent in an operations log for recovery. The 3rd type is obtained by querying DataNodes at startup and periodically thereafter.

Reading in HDFS

When reading data, the client goes to the master node to request the location of the data of interet. The master node returns this information and the application then goes directly to the slave/storage node.

Block replication

The master node replicates blocks to ensure data reliability and availability. Furthermore, it is done for load balancing.

Week 9 - Distributed Data Processing

Distributed Data Processing

Motivation

We are interested in big data processing by using parallelisms on a single node in a multicore machine. We want to tune the performance of the system and how they work internally.

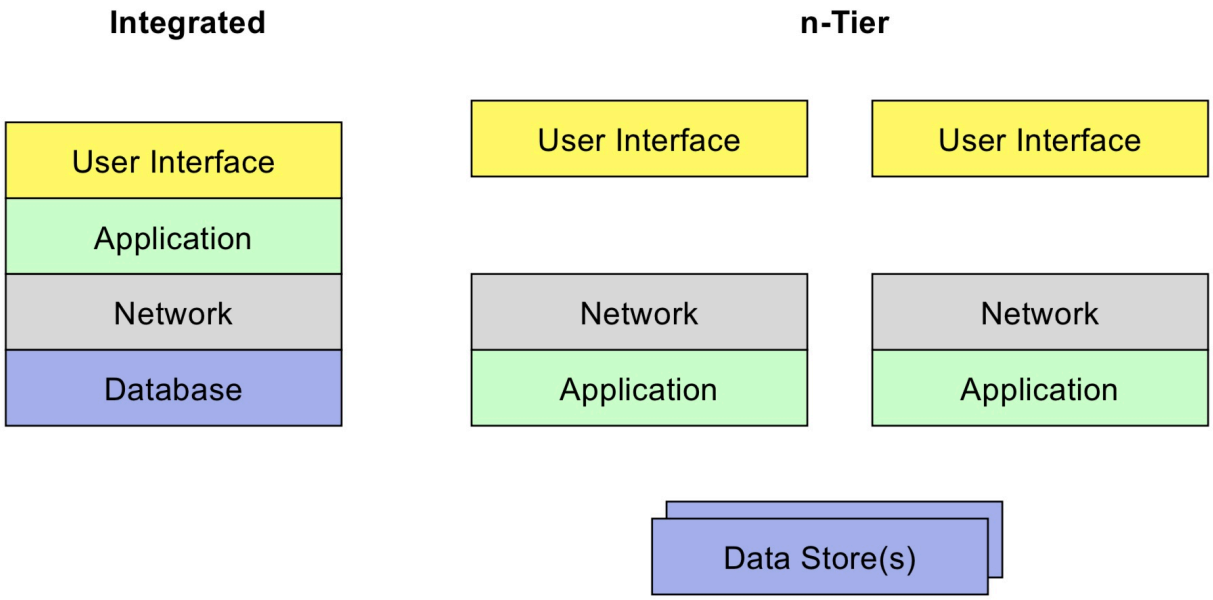
Web-Scale Data Managment

Main challenges:

- How to store and share data in cloud?
- How to process this large data

Scale-up vs Scale-out

We need a n-distributed architecture for webapp to be able to handle data



Scale-Up

Buying larger hardware. You can buy large powerful hardware for this. These are powerful and requires alot of power and cash to purchase.

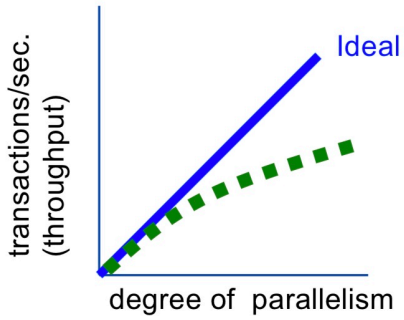
Scale-Out

This is a cheaper way to scale.

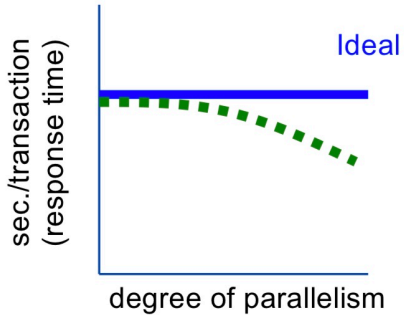
The goals of scalability is to grow with the load.

The Goals of Scalability

- Speed-Up
  - More resources means proportionally less time for given amount of data.
    - but: coordination overhead



- Scale-Up
  - If resources increased in proportion to increase in data size, response time should remain constant
    - again: modulo overhead



Advantages of scaling out:

- Low initial costs
- Incremental scalability with load
- We can just purchase cheap commodity hardware

Disadvantages of scaling out:

- Need software to handle scalability and consistency
- It can be difficult to guarantee availability

2 Goals for distributed data processing:

1. Scale-Agnostic Data Management

- Sharding for performance
- Replication for availability

2. Scale-Agnostic Data Processing

- Need to be able to parallelize across lots of CPUs
- We want the system to always be available
- We want the system to be elastic and scale up/down according to needs.

Distributed Join Processing

- Distributed join algorithms split the joining of pair-of-tuples over several nodes, each computing part of the join locally. To produce the final result, the local results of each node are collected together.
- **Approach 1: Local (replicated) reference tables**
  - If one of the joined tables is small and stable enough, replicate (copy) it to all cluster nodes
  - Then local join per node possible with any of the discussed join algorithms
- **Approach 2: Broadcast Join**
  - If one join table has a filter condition which reduces it to a small-enough size, broadcast filtered tuples to all nodes, with subsequent local join per node.
- **Approach 3: Distributed-Shuffle Join**
  - No assumption on joined data, but needs Equi-Join or Natural-Join
  - State-of-the-Art: Distributed Parallel Hash Join
  - Shuffling/Sorting of input data, then re-distribute to 1 node based on partitioning
- **Approach 4: Fragment-and-Replicate Join**
  - Fragment both input relations, then replicate fragments to multiple join nodes
  - Works with any join condition

1) Local Replicated/Reference Tables

Assume that table S is much smaller than table R. Table R is partitioned over all n nodes whilst table S is replicated on all n nodes. We then do a local join on each site and merge the results back.

2) Broadcast Join

We assume that table S is very small or has a selective filter predicate on it. Table R is partitioned over all n nodes. Table S is then either partitioned as well or only resides on a single node. Broadcast filtered data into all nodes. Then perform local joins on each node.

3) Distributed-Shuffle Join

This is only for equi-join or natural joins. Table R and S are not partitioned yet. We shuffle and partition data in the same manner by either a rangoe or hash partition on the join attribute across all the nodes. We then run local joins on each node.

Distributed Parallel Hash-Join: Special case of distributed shuffle join. Here, we use 2 hash functions. We assume S is smaller than R. We use hash function h_1 to take the join attribute value of each tuple in S and map to the node. Each node N_i reads the tuples of S on its disk D_i and sends each tuple to appropriate node based on hash function h_1. We then use another hash function h_2 to help compute the hash-join locally. We then take the larger relation R and resdistribute them across n nodes using hash function h_1. We then repartition the R tuples at the destination node with hash function h_2. Each node N_i then hash joins the tuples.

4) Fragment-And-Replicate Join

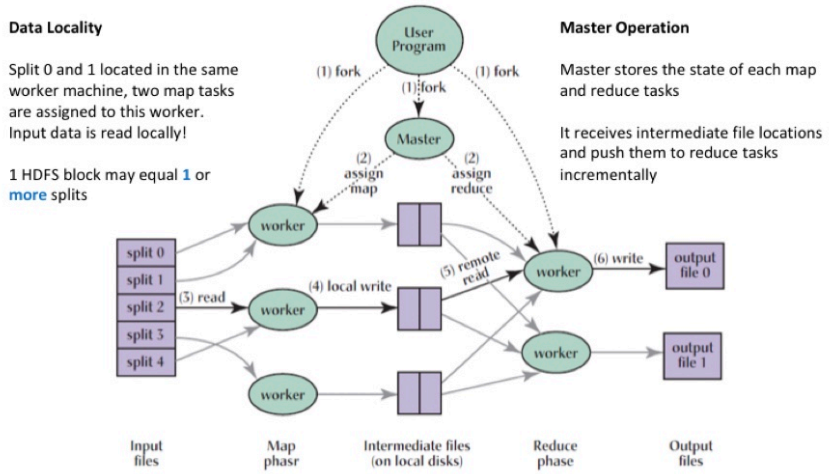
This is general purpose join. This is for non-equijoin scenarios. Partition data based on hashing (or range). Replicate this data cross nodes. Broadcast join is actually a case of an asymmetric fragment-and-replicate join. So we partition R into **n partitions** and S into **m partitions**. We require **m*n** nodes where each node P_{ij} local joins partition i of table R and partition j of table S. We do local joins on each node processor.# Scale-Agnostic Data Processing

MapReduce

- Declarative

1. Scan large volume data
2. Map: Extract some interesting information
3. Shuffle and sort intermediate results
4. Reduce: Aggergate intermediate results
5. Generate final output

We want to abstract these two operation by using functional programming ideas. "Stateless" and no "side effects". We can do everything locally. If a node fails, it doesn't affect other nodes since no side effects and stateless.



The framework will shuffle the data for us. We only need to worry about the map and reduce implementation. The only important part is figuring out what the arguments and return data of the functions.

Map-Reduce job is sub-divided into several tasks. A scheduler will assign tasks to machines.

MapReduce vs SQL

MapReduce is similar to a parallelized aggregation query. We have control of the WHERE and AGG part

Note the similarities to a (parallelized) aggregation query:

- **SELECT** AGG(data)
  FROM DataSet
  WHERE condition
  GROUP BY attr **ORDER BY** attr

Data is partitioned so we can work with it faster

MapReduce means we have to manually code alot of things at a time for each

Characteristic of Map Reduce

- Input data partitioned so each worker only has to process part of the data in parallel
- Materialization approach to execution

- Grouping/partitioning is done using sorting and merging
- It is optimised for a shared nothing architecture

Issues with MapReduce

1. Doing joins and lots of other things are very tedious and verbose since only map and reduce command.
2. Wrong level of abstraction for non-programmers.

How to write joins with MapReduce:

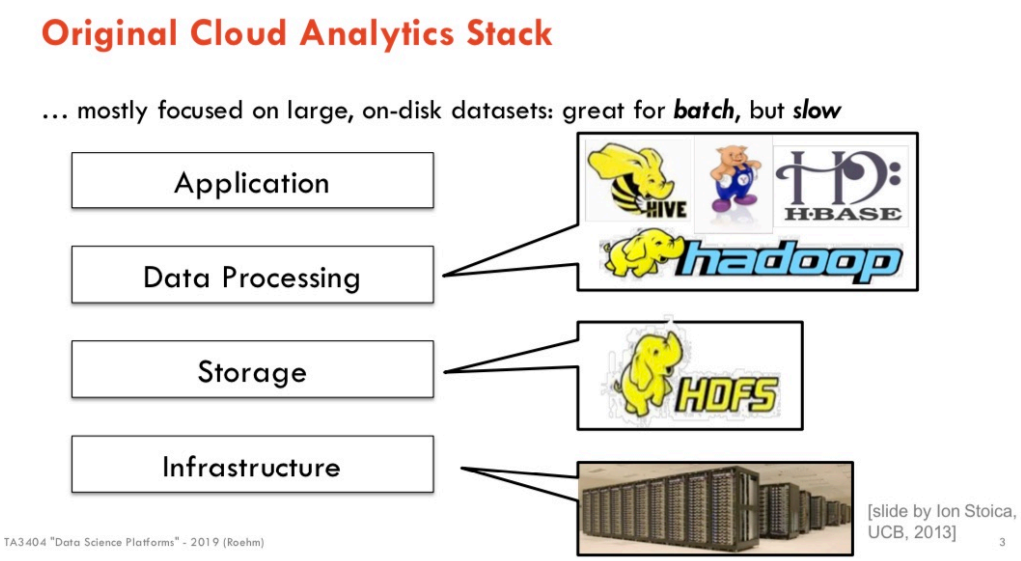
1. Broadcast join. Load data into distributed cache and replicate data into the memory or local file system. We can write a Join-Mapper function that locally joins each tuple with every tuple from local copy of DistributedCache.
2. Partitioned Join. We load in data to be joined and map them by the join key attribute. This only works for Equi-join or Natural join.

Week 10 - Distributed Data Flow Platforms

Motivation

We are now looking towards alternative platforms for data processing.

Hadoop World



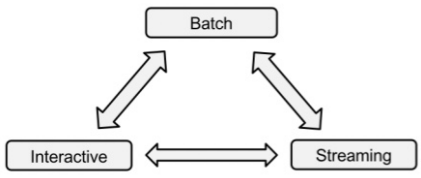
- Infrastructure: Parallel infrastructure. Share nothing and scale out.
- Storage: HDFS handles data by data partitioning and data replication for access and dealing with access. It helps to abstract away on how storage occurs.
- Data Processing: Hadoop processes your data with a combination of Map and Reduce operations.
- Application: Frameworks to work with the analytics.

Map/Reduce

Map/Reduce allows simple parallelization of pipelined tasks.

Pros: Parallelism, fault-tolerance, runtime decides where to run tasks.
Cons: Simple processing model where data flows from table storage to stable storage + materialization.

Map/reduce will work with data from a materializiation perspective. Bad for doing expressive and more complex tasks.



We want to see how we can combine the different computations between interactive/batch/streaming. Hence, we need more sophisticated algorithms.

Alternative approaches we can use for better processing:

- We use more work with main memory. Cache the data and use it more extensively. Memory is cheap nowadays and keep the data in a pipeline.
- Use acyclic data flow plans with advanced data-flow operators to help with the dataflow. This give us more options on how to process our data.
- Get more stuff out of the share nothing architecture by increasing the parallelism of the data.
- Meanwhile we should use the current existing infrastructure such as Hadoop and HDFS to help us process data.

Apache Spark

Main idea: Materialize into main memory as much as possible and materialize at the last moment.

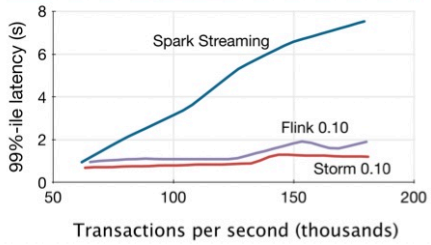
Resilient Distributed Dataset (RDD). Fault-tolerance and in-memory storage abstraction. We don't care where the data is stored. We have few operations we can apply on the dataset which the programmer does not need to implement. Instead of materializing the data, we materialise the logical data steps that was taken.

We can transform data in stable storage using data flow operators such as (map, filter, group-by etc). Hence, we have alot of abstractions on how we can now process the data of interest.

Apache Flink

Apache Flink: Dataflow-oriented distributed processing engine. It has more powerful data flow operators and optimisation routines. It follows a pipelined execution model with an automated plan optimisation. It focuses more on streaming data compared to Spark.

	Apache Spark	Apache Flink
Principle	set-oriented data transformations in stages	transformations by iterating over collections with pipelining
Data Abstraction	RDD	DataSet
Processing Stages	separate stages	overlapping stages
Optimiser	with SparkSQL	integrated into API
Batch Processing	RDD	DataSet
Stream Processing	micro-batching	pipelining; DataStream API

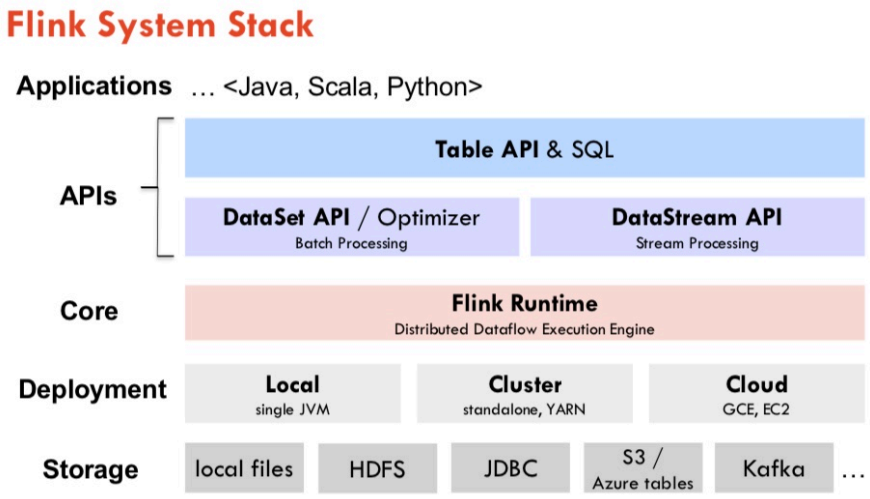


Core features:

- DataSet API: Batch processing/fixed datasets
- Datastream API: For streaming datasets
- Table API: Doesn't matter if batch or continuous. It is an abstraction on top of dataset/datastream API.
- It has a Built-In optimiser, in particular, it is a cost-based optimiser which helps with deciding the executing strategy based on inputs.
- Data transformations supporting UDFs
- In-memory pipelining: Efficient stream processing
- Iterative algorithms: Repeat operations

- Lazy evaluation

It has a built-in optimizer to help optimize the query.



Numerous sources for data storage. Deployment can be local or standalone.

Assignment uses standalone for SIT lab machine. Single machine with multiple processes

Distributed dataflow execution engine: Handles dataflow between operation

YARN Resource Manager

YARN's main goal is to act a scheduler by arbitrating available resources in the system to competing applications.

Distributed processing frameworks rely on resource managers to deploy on clusters. YARN helps to allocate work to be done on different nodes. In particular, it focuses on storage, memory, and computation. The client asks YARN which worker nodes can we utilise to do work.

On top of this, we have node managers (one for each node) to manage resources for a given node and application managers (one per application) to request resources from the cluster on behalf of an application. It is used to monitor the execution and progress of the application.

Flink Runtime Internals

Flink: You specify code and optimizer does its work. Then it coordinates with the YARN to help distribute the work with the TaskManagers.

Flink Data Model

The flink data model has a collection of tuples that are heterogenous type. Data does not have to be primitive. When its not, then we have JSON.

Flink Data Transformation

Data Transformation:

- 2 components: User-defined function and parallel operator function
- Second part is what parallelises the UDF

Map/FlatMap

- Flatmap: We get n words of output
- Map: Only 1 output

Reduce:

- Input organised by flink framework which in this case is by the key. The UDF can then reduce it using this key.

An optimisiation for the reduce operation is combine.

Iterate operator: Simple form of iteration where each step function consumes the entire input

Join Strategies in Flink

There are 2 distributed join strategies:

- Repartition Repartition Strategy (RR): Distributed Shuffle join. Takes in 2 tables and partitions them equally across each node and then perform local joins.
- Broadcast-Forward Strategy (BR): We choose one table to get replicated and broadcasted to all the worker nodes. The other input relation stay where it is.

Local join algorithms in Flink on the actual node:

- **Sort-Merge Join**: Similar to what we have seen earlier in the course.
- **Hybrid-Hash-Join**: Spills some of the hash table and probe function into disk if data is too large.

Other facts on Flink

Flink Optimizer:
Flink has a dedicated optimiser to help figure out
an efficient execution plan for data processing task.

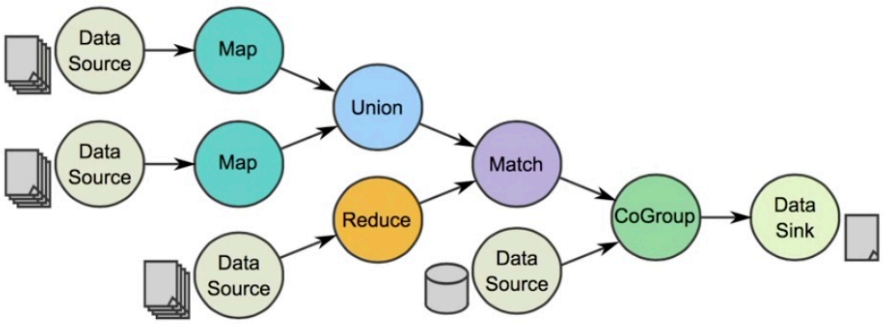
Flink does memory handling very well. Flink represents data in the raw byte form.

DataFlow in Flink

In Flink, the data flow is defined in the form of an operator DAG that consists of

1. Data Sources - Entry point of data into data flow. Can connect to databases etc and produces exactly one dataset.
2. Data Transformations
3. Data Sinks - These are the exit points where data leaves the data flow. Output formats can include things such as CSV Files/binary files etc.

Below is an example of how data flows in Flink.


Data Source
Map
Data Source
Map
Union
Match
Reduce
CoGroup
Data Sink
Data Source
Data Source

Week 11 - Data Stream Processing

Stream Query Processing Principles

Motivation

Applications involving streamed data needs to be processed in real time.

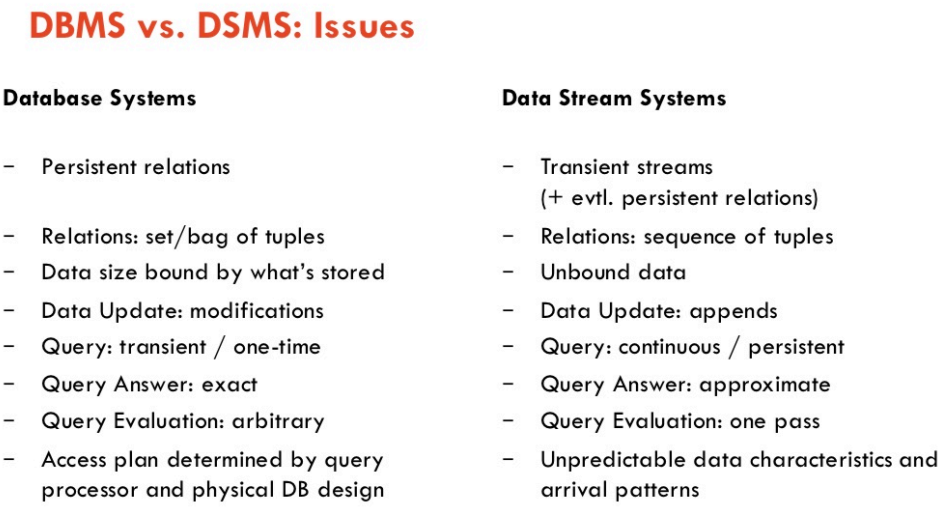
We don't use DBMS as it is not agile enough. We want be able to process data as it flows without storing it persistently. Loading in data is annoying and slow. Processing queries are meant for running on whole dataset, we want real time querying. DBMS tends to prefer data that does not require frequent updates.

Stream processing tries to process slightly buffered data, so we have windows in which to do operations on. In streaming, we run query on input data, combine with previous state, and store.

2 application areas of streaming data:

1. Transaction Data: Payments. Log interaction between entitites.
2. Measurement data: Networking traffic, sensors, etc. Keep track of entities.

Query first then storage rather than storage then query!!!



Definition (Data Stream). A data stream is a sequence (implicit order) of tuples. The stream is potentially unbounded. Updates are usually appends rather than inserts. We continuously run queries, so we have an approximate result.

Data Stream Processing

Streaming operators can be stateless or stateful. **Stateless** operator works at each event individually whilst **stateful** operator works on aggregated output.

To get stateful operators to work, we can use the *windowing technique*.

Windowing Technique

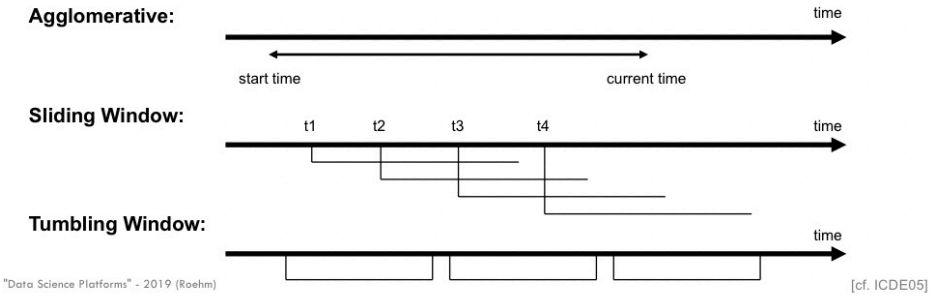
Window: Extracting a finite relation from an infinite stream. There are many ways of defining windows:

1. Window based on an attribute like time
2. Use tuple counts
3. Use explicit markers such as punctuations (e.g. close of an auction in financial data)

Ordering-Attribute-Based Windows

Here we assume we have an attribute to define order of stream. Usually it is the time attribute. We can have variations of this such as:

1. Agglomerative: Expanding window from a start time.
2. Sliding window: Every 10 min, start a new window.
3. Tumbling window: Smaller than sliding window. Moving average every hour for 30 minutes.



Time is not quite clear in this scenario.

Notion of time in attribute based windows

Event Time: The time for event to be processed.

Processing Time: Time for when the event is actually processed. e.g. ingestion time.

Star War episodes example:
Event time: In the universe of when star war events actually happened.
Processing time: When the actual movie was released and shown.

Count-Based Window

Count number of tuples to analyse. We have start and ending points for each window. We can have sliding or tumbling over the stream. The last N values does not mean the last N values from a logical point of view.

Punctuation-based window:

Look at actual content in the stream. There are markers to help demarcate when each event occurs in the stream. An example would be a stream of auctions.

Challenges with streaming data

- Data is unbounded
- Out of order processing
- Stream rates may be unpredictable and vary alot. Hence we can overload the system or see conditions change.

Streaming Processing Systems

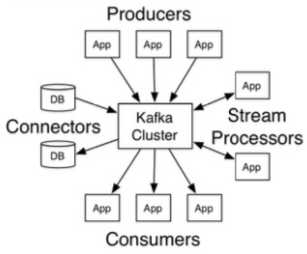
Kafka

Kafka is a distrbuted publish-subscribe messaging system. It can also be thought of as a **stream-oriented messaging system.** Here, there is a separation of handling data stream and processing the data itself. Kafka just handles streams, doesn't know what to actually do with it.

The workflow involving Kafka is as follows:

1. Producers of streaming events (stockmarket api, sensors etc)
2. We have consumers watching the streams (could be stream processing systems or databases)
3. The producer publishes streams of data and consumers subscribes to the stream.

Kafka negotiates how produces publishes streams without knowing who will receive it whilst handles how consumers subscribe to data without worrying who is it actually from.



3 capabilities of Kafka:

1. Publish and subscribe to streams of records
2. Store streams of data in fault-tolerant manner
3. Let apps process streams of records as they occur

Streams can be thought of as unbounded log. Producers write/append onto our log whilst consumers will read in the log. Hence, Kafka maintains a partitioned log for each topic, which is replicated across multiple servers. Topics in Kafka are multi-subscriber, so each record published to a topic is delivered to one consumer instance with each subscribing consumer group.

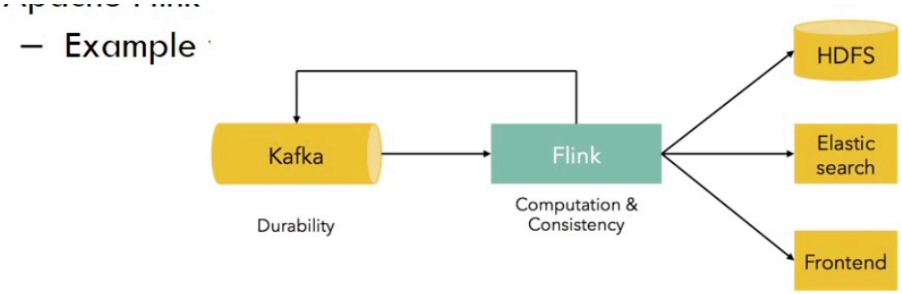
Kafka as Storage System and guarantees

The log is replicated aross multiple servers. We don't just write to one node, we write to multiple nodes so we don't go any system failures. We don't overload a particular node. We have partitioning and replication in Kafka. Kafka can also store data. It logs to disk. Kafka has to let the producers know when it is done writing the topic published to a disk so that the producer knows whether does it have to repeat the information again.

Every message going into Kafka is processed at least once. This is so that if a node goes down, we get data from another node. However, that data could be something processed before. Kafka chooses having repetitive data rather than losing any data.

Kafka + Flink

Kafka is not a data stream processor. It only serves as messaging and storage system for data streams. Flink/Spark/Storm takes in input from Kafka and outputs to a file system/web app.



The datastream API in Flink can be used to subscribe to Kafka. There is fixed, sliding, session windows for Apache flink.

3 types of time in Flink:

1. Event time. Semantics for the actual source itself. e.g. when Star movie was set in
2. Ingestion time: When Flink ingested the data from Kafka
3. Window Processing Time: When does window occurr in our processing.

We have access to all 3 in Flink.

Apache Flink does alot better than Spark in terms of handling transactions per second.

Week 12 - NoSQL Databases

No-SQL Database

Relational DBMS: Fixed structure tuples, 1NF, joins are important since in 1NF, SQL on top, and are focused on transactions.

NoSQL: Document-oriented databases, No-SQL column stores (no transactions now), pure key value: tuples are key value so quick to query and graph database.

Relational tend to run on single server but these NoSQL will be more focused on distributed system. Furthermore, these NoSQL setups do not need to define schema beforehand and also have more modifications on the data structure of which the data is stored as.

NoSQL Classification Attempt

	SQL	Document Stores	Column Stores	Key-Value Stores
Data Model	set of tuples	set of nested/rich named fields	(key,column,value) triples	(key, value) pairs
Schema	schema first	schema on read	data-defined schema	schema-less
Querying	SQL	selections on keys	selections on keys, including scans	select key
Updates	in-place update of single or tuple set	in-place update of single field or doc	via creation of new column values	update(key: value)
Transactions	ACID	None (* -> v4)	None	None; assumes single (k,v) access
Physical Design	Indexes, Partition, Mat. Views, …	some systems with secondary indexes	Index on key	Index on key
Distribution	horizontal&vertical partitioning	horizontal partitioning	horizontal and vertical partitioning	partitioning by key (horizontal)
Replication	all kinds…	eventual consistency	HDFS replication	eventual consistency
Examples	Postgres,MySQL, Oracle, DB2, SQL Server, Sybase	MongoDB Apache CouchDB	HBase, BigTable	Amazon Dynamo Cassandra

In NoSQL, there is

- No concurrency controls
- No recovery managers or at least limited to one node
- No schema management
- Alot of query evaluation isn't available. They are more focused on being a storage manager.

Key-Value Stores

Amazon Dynamo

- Key-value store. Only 2 operations: put and get. The put method has an additional parameter of context involved

- We have fast persistent data storage
- It assumes unique ids and inputs are independent of each other
- Require small images (no images/movies)
- Very scalable/fast/reliable systems

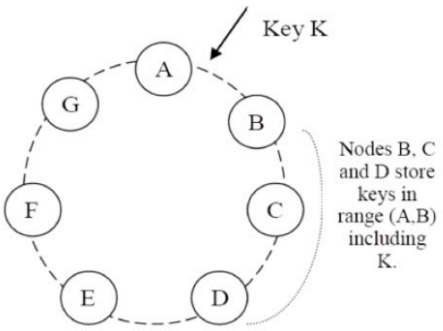
Partitioning in dynamo
We store values as well in subsequent nodes (N nodes after) so we can recover information. If our hash hashes to a key, and there is nothing there, it snaps to the closest node and get corresponding entries. Less nodes in the system, the more data you have to store for a wider range of hash values. So if we update something, we need to replicate N times for each one.

- Dynamo is a **distributed hash table** with replication

Put(key, contex, object)
Get(key)

Context: Vector clock/history information.

No guaranteed that updates are consistent. Get() may return different version of same object.



Inside we have a hash table as a ring. We get a key from our hash value. So dynamo is tolerable for node failures.

We prioritise high availability and tolerance to partitioning in the system. However, we lost consistency from the CAP system. We need to handle consistency in the application.

CAP for Dynamo: Prioritise availability and partitions.

Vector clock: Array of pairs of values. In particular, it involves the node and counter value.
Quorum Protocol: Used for the get() and put() method. Here, the issue is that Dynamo utilises asynchronous replication so we may be getting back inconsistent values in the system. Hence the Quorum protocol is a weighted voting system which requires a certain minimum number of nodes to participate in a successful read/write operation.

Summary of Techniques in Dynamo

Problem	Technique	Expected Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Cassandra

Apachae Cassandra is an open-source distributed storage system developed at Facebook. It is an open source version of Dynamo. It implements alot of similar ideas as seen in dynamos except

- It has (key,value) pairs where values can have a nested sub-structure.
- value has several addressable columns which can be grouped into column-families.

It follows the key ideas of Dynamo which are:

- Paritioning with a hash-based ring topology
- Replication with a N=3 replication factor using quorum-based operations
- Cluster management handled using gossip-based communication protocols.

Column Stores

HBase

HBase is more of a data store rather than database. Google's Bigtable started NoSQL industry. Led to HBase, Cassandra and other NoSQL databases.
Built on top of HDFS system. It is used to retrieve data quickly. HBase is a **column family oriented database. **

HBase can be seen as big tables made up of rows and columns where every row has a unique row key. There is an infinite number of columns and each row does not need a value for every column.

The HTable comprises of a (row-key, column, timestamp) triples as the basis for lookup, inserts and deletes.

HBase is a column store where tables are stored in a per-column-family fashion. Tables are then partitioned into regions where each region is made up of multiple stores.

MongoDB

MongoDB is a docuemnt-oriented storage with a flexible schema. It is a collection of documents with nested key-value pairs based on JSON format. Very popular for object oriented software development model as we can match JSON documents to objects. There is no SQL query language, only an API.

The properties of this document format are:

- Polymorphic: Docuemnts in a collection can contain different fields.
- Dynamic: Self-describing data needs no fixed schema.
- Govern: We have a JSON schema to enforce structure.

Relational Database	MongoDB
Database	Database
Table	Collection
Row (tuple, record)	Document
Column (attribute)	Field
Schema First; tuples in a relation follow the same schema	Flexible schema; documents in a collection do not need to have the same set of fields
Normalised Schema; 1NF	Denormalised data model: documents can be nested

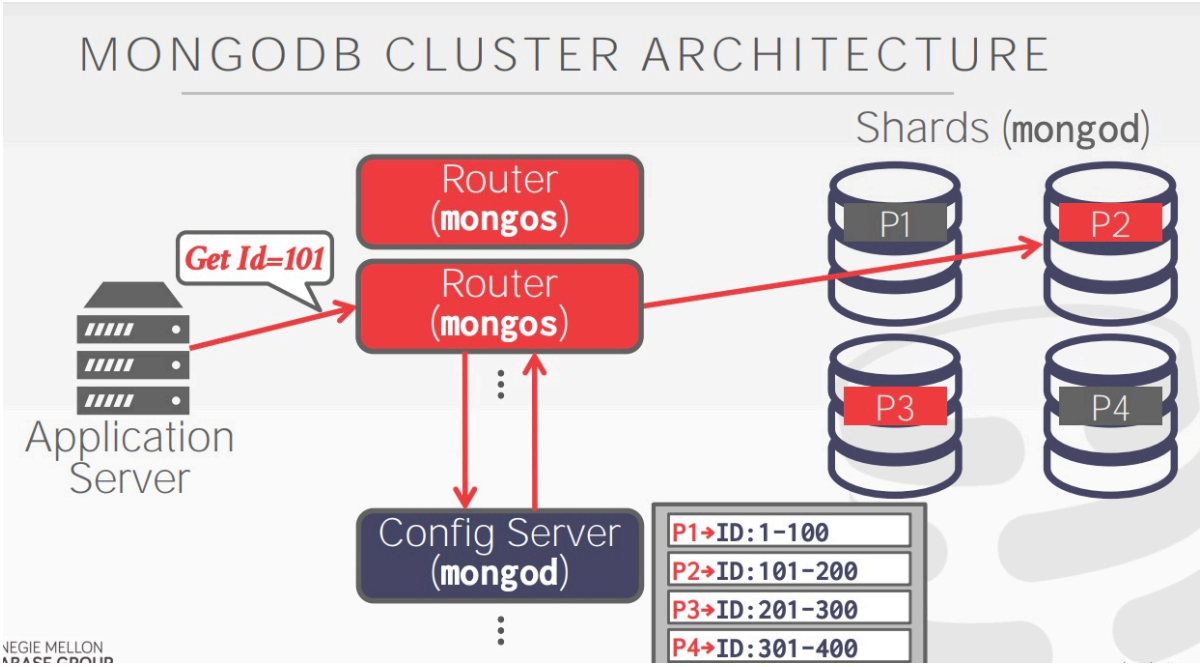
Features of MongoDB

- Automatic Data Partitioning: It auto shards large collections among multiple machines in order-preserving manner. Recall that this means it does horizontal range partitioning per table.
- Master slave replication: There is an asynchronous master-slave replication with one shard. This allows for availability.
- Handles load balancing.
- You can also specify how many replicas to set up, which in generally is usually set to 3 replicas
- MongoDB has a data model and supports secondary indexes such as B+ trees, which can be used on any attribute. Furthermore, it can also support multi-column indexes too.

MongoDB Cluster Architecture

Heterogenous architecture.

1. Router node (Mongos)
2. Config server node (mongod)
3. Shards node (Mongod)



- Application send query to router and access for id. Router is stateless.

- Router sends query to config server which has its own partitioning table and figure out where id is. The config server returns where the id is located back to the router
- The router now knows where the data is stored in the shard

You can scale all these things out now. The issue with this is that you have to do alot of unnecessary communications.