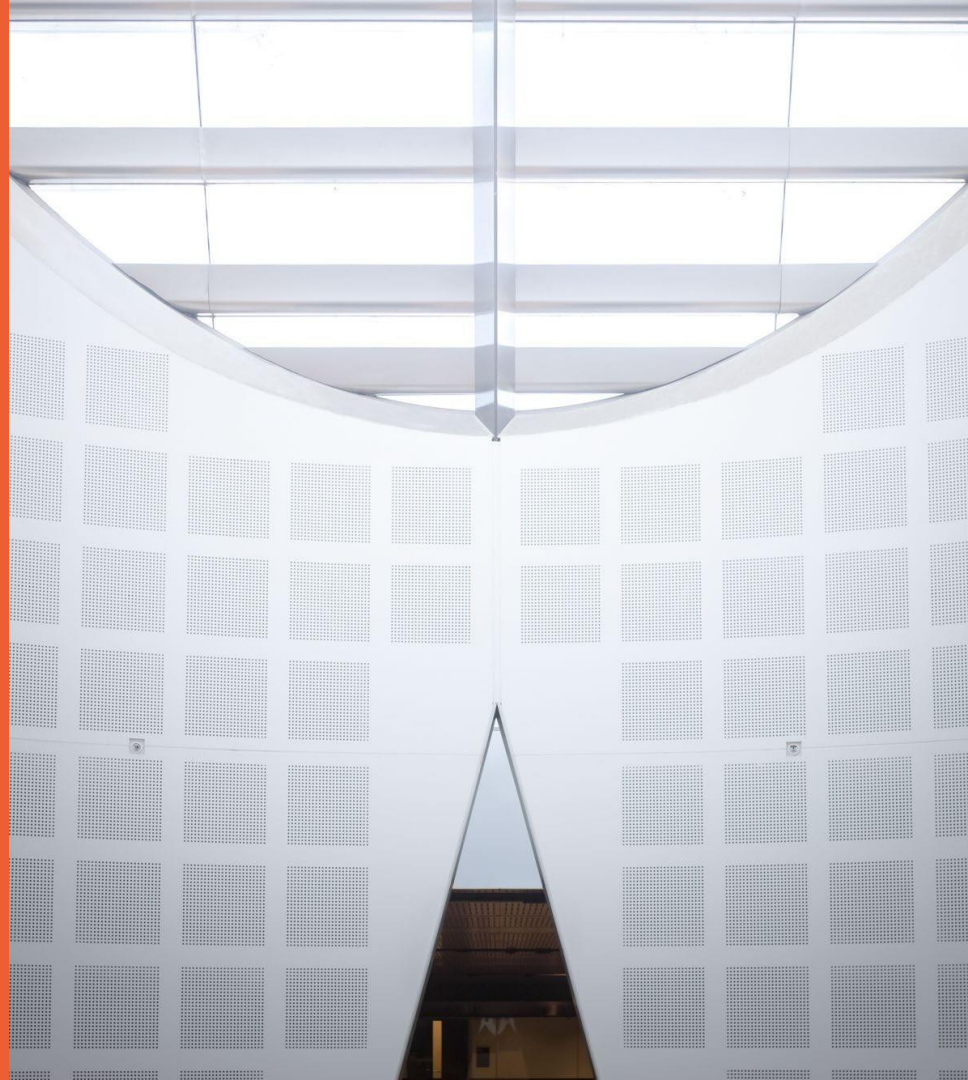
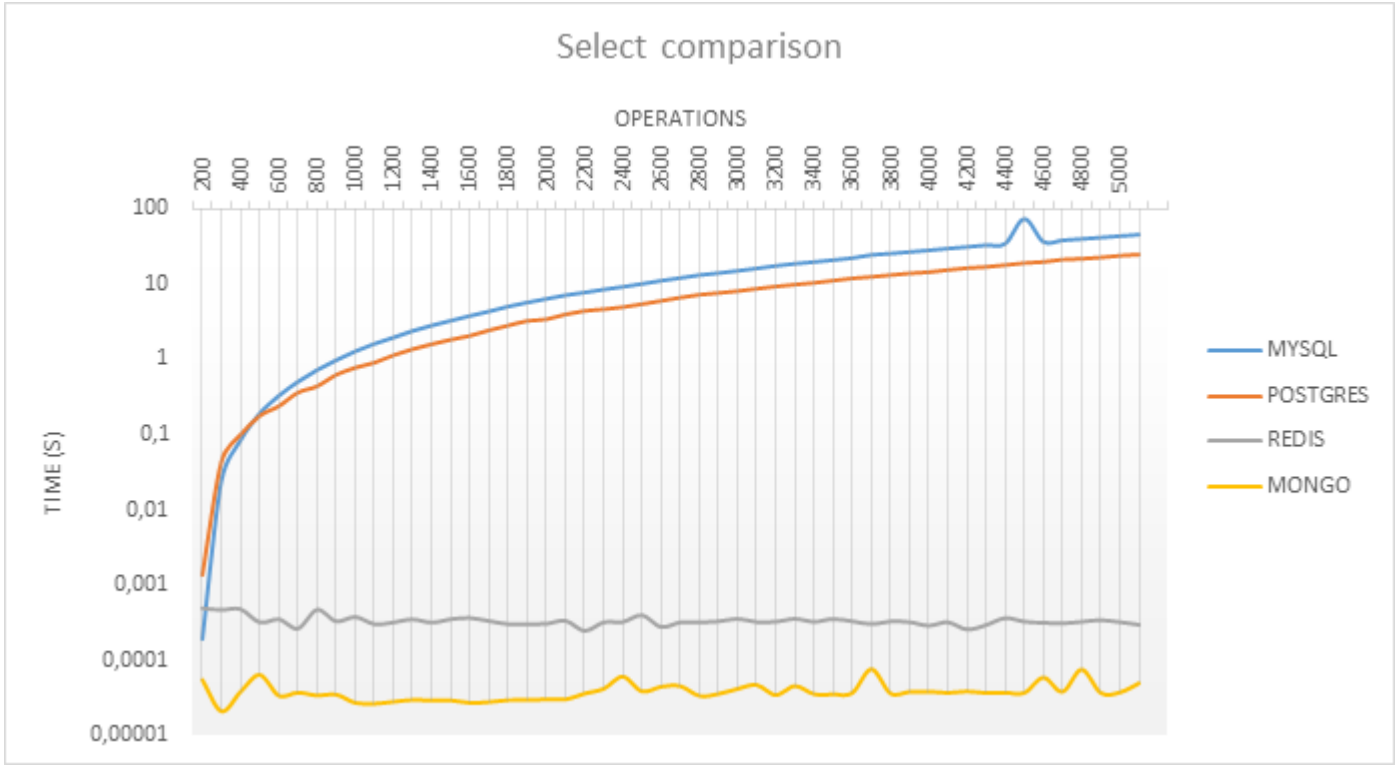
DATA2001: Data Science, Big Data and Data Diversity

Week 5:
Scalable Data Analytics
The Role of Indexes and Data Partitioning

Dr. Nasim Ahmed
School of Computer Science

THE UNIVERSITY OF SYDNEY





https://profil-software.com/blog/development/database-comparison-sql-vs-nosql-mysql-vs-postgresql-vs-redis-vs-mongodb/

https://dev.to/profilsoftware/database-comparison-sql-vs-nosql-mysql-vs-postgresql-vs-redis-vs-mongodb-2e0l

Comparing DBMS https://www.altexsoft.com/blog/comparing-database-management-systems-mysql-postgresql-mssql-server-mongodb-elasticsearch-and-others/

This Week's Learning Objectives

- **Data Storage Layer:**
  - Motivation: Data stored on disks
- **Indexing**
  - Efficient data access based on *search keys*
  - Several design decisions…
  - Awareness of the trade-off between query performance and indexing costs (updates)
- **Distributed Data Management**
  - Data Partitioning / Sharding
  - Data Replication

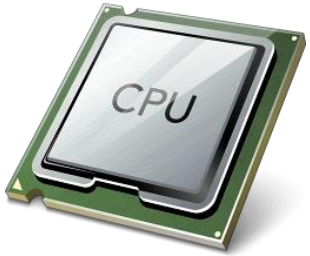
Based on slides from Kifer/Bernstein/Lewis (2006) "Database Systems" and from Ramakrishnan/Gehrke (2003) "Database Management Systems", and including material from Fekete and Roehm

Data Storage

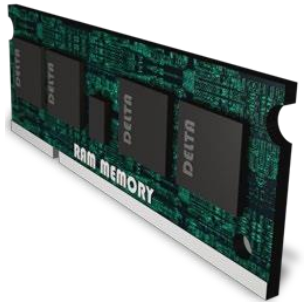
Where is Data Stored?

Main Memory (RAM):
- Expensive
- Volatile

Secondary Storage (HDD):
- Cheap
- Stable
- BIG



FAST

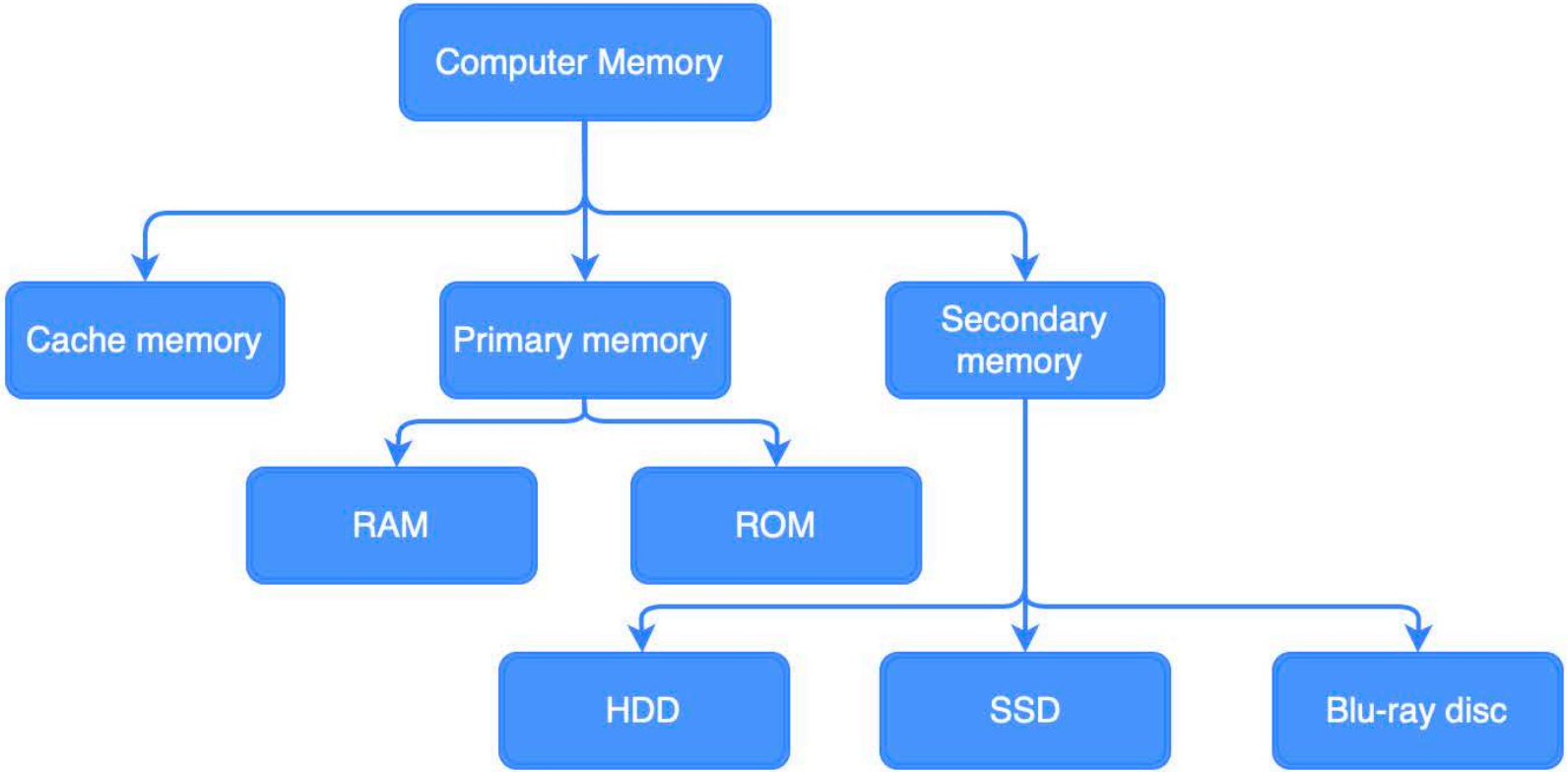


SLOW

IBM Hybrid cloud

Tertiary Storage (e.g. Tape):
- Very Cheap
- Stable

Data Storage Classification



Memory Classification

Data Storage

- Data must be stored on disks.
  - *Main memory is more expensive than HDDs,* (Amazon live disk price) *so you can't have very much of it.*

Price per TB	Price	Capacity	Technology	Condition
A$67.50	A$135	2 TB	SSD	New
A$28.62	A$229	8 TB	HDD	New

  - *Main memory is volatile.*

    We want data to be saved between runs. (Obviously!)

- This has major implications for data science platform design!
  - **READ:** transfer data from disk to main memory (RAM).
  - **WRITE:** transfer data from RAM to disk.
  - Both are high-cost operations, relative to in-memory operations (HDDs are about 100,000x slower than main memory access), so must be planned carefully!

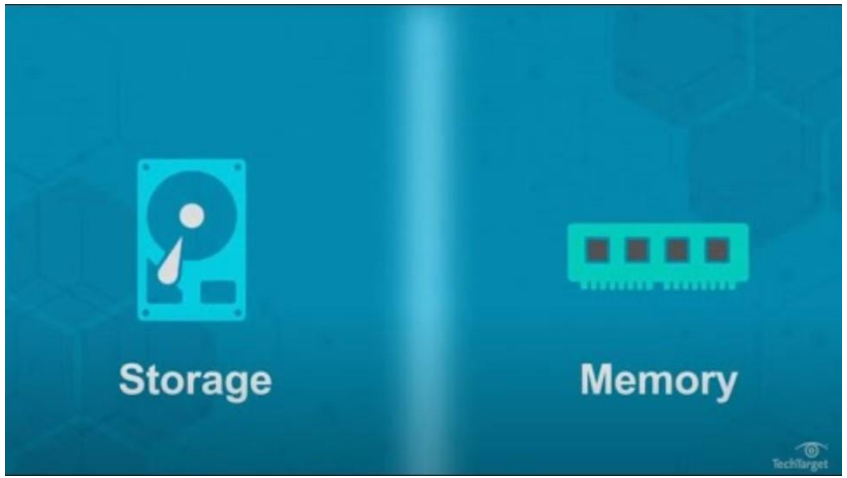
Data Storage

HDD can be about 100,000× slower than DRAM

Main Memory (DRAM) latency: ~100 ns

HDD latency (random access): ~10 ms = 10,000,000 ns

$$\frac{10,000,000 \text{ ns (HDD)}}{100 \text{ ns (DRAM)}} = 100{,}000$$



https://www.youtube.com/watch?v=H_M--weEzpA&ab_channel=EyeonTech

Solid State Drive (SSD)

- Pro:
  - SSDs have about the same access characteristics for random or seq. access
  - About 100x faster than HDDs
  - Higher IOPs rate than HDDs (IOP: Input/Output Operations)

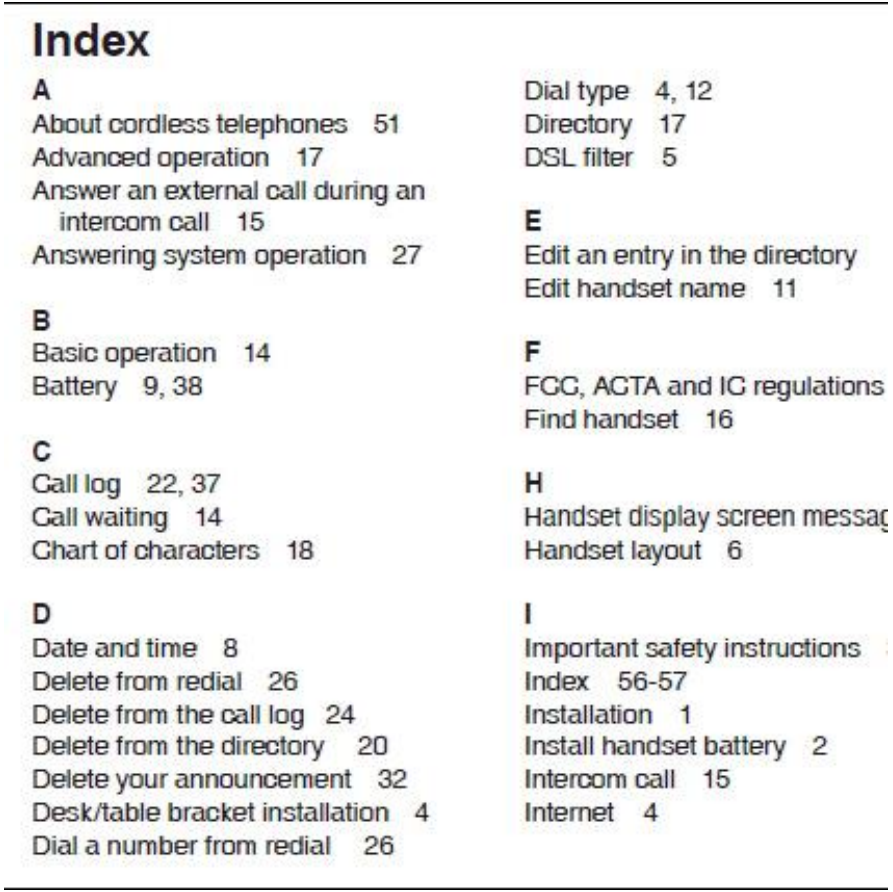
- Cons:
  - SSDs 4-6 times more expensive than HDDs
  - 100x faster than HDDs still leaves us 100x times slower than main mem

How to Access Data on Secondary Storage FAST?

- Key Challenge: Secondary storage needed for sheer data volume (and persistence), but it is slow…

- Approaches:
  - **Block-wise transfer**
    - transfer data in fixed-size chunks (blocks or pages) between storage layers
  - **Caching / Buffering**
    - Keep 'hot' data in memory, use secondary storage for 'cold' data
  - Optimised File Organisation
    - Heap Files vs. **Sorted Files;** Row Stores vs **Column Stores**
  - **Indexing**
  - **Partitioning**

Indexing



Index

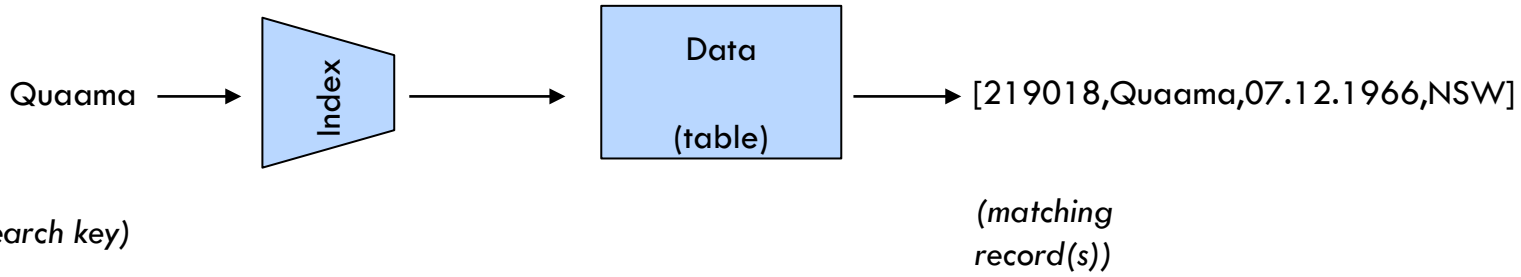
What is Indexing

a data structure technique to efficiently retrieve records from the database files .

- Idea: Separate location mechanism from data storage
  - Just remember a book index:

    Index is a set of pages (a separate file) with pointers (page numbers) to the data page which contains the value
  - Instead of scanning through whole book (relation) each time, using the index is much faster to navigate (less data to search)
  - Index typically much smaller than the actual data

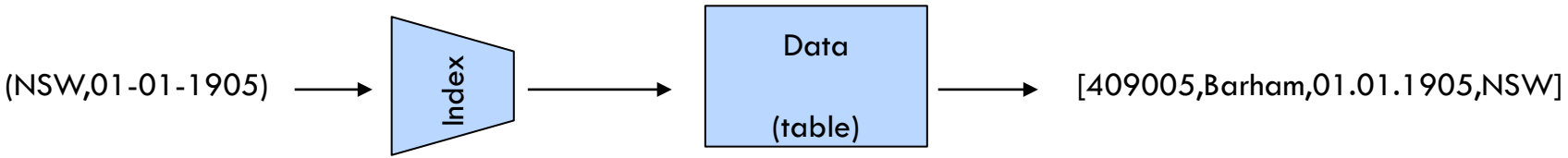
Index Example



- Here, index is on name attribute of Stations table
- We say name is the *search key* for this index (it is the attribute which we use to look up data)

Multi-attribute index

- The search key in an index can be a combination of two or more columns, not just a single column
- Eg create index on (state, commence) for Stations table
  - Here, we can use this to retrieve all rows where state='NSW' and commence = 01-01-1905



Index Definition in SQL

- Create an index

  CREATE INDEX ***name*** **ON** ***relation-name*** **(<*attributelist*>)**

  - Example:

    CREATE INDEX *StationNameIdx* **ON** Stations(*name*)

- Index on primary key generally created automatically
  - Use **CREATE UNIQUE INDEX** to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- Indexes are used automatically by a database – no way explicitly use

- To drop an index

  DROP INDEX ***index-name***

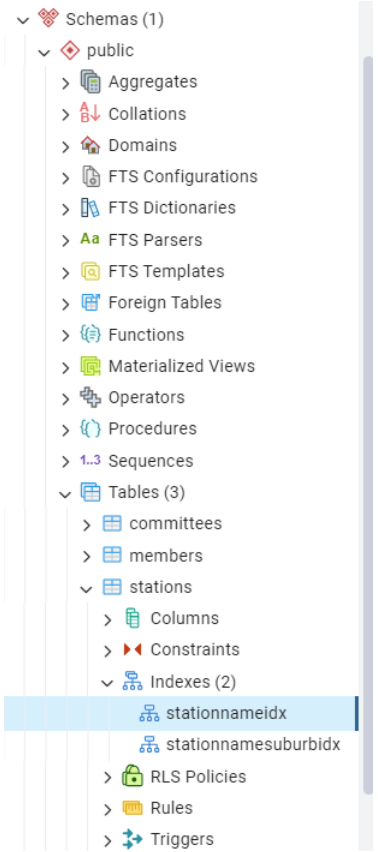
Example

Suppose that search queries on the stations table are very slow and we want to use an index on the “Station Name" to speed up the queries, we can use the following script to achieve that.

```
CREATE TABLE stations (
    ServiceStationNo INTEGER,
    ServiceStationName VARCHAR(100),
    Address VARCHAR(1000),
    Suburb VARCHAR(100),
    Postcode INTEGER,
    Company VARCHAR(100),
    PRIMARY KEY (ServiceStationNo)
);
```

```
Create index StationnameIdx on stations (ServiceStationName)

Create index StationnamesuburbIdx on stations (ServiceStationName,Suburb)
```



Indexes - The Downside (HW)

- Additional I/O to access index pages
  (except if index is small enough to fit in main memory)
  - The hope is that this is less than the saving of more efficient finding of data records
- Index must be updated when table is modified.
  - depends on index structure, but in general can become quite costly
  - so every additional index makes update slower…

- Decisions, decisions…
  - Index on primary key is generally created automatically
  - Other indices must be defined by DBA or user, through vendor specific statements
  - Choose which indices are worthwhile, based on workload of queries
    (cf. later this lecture)

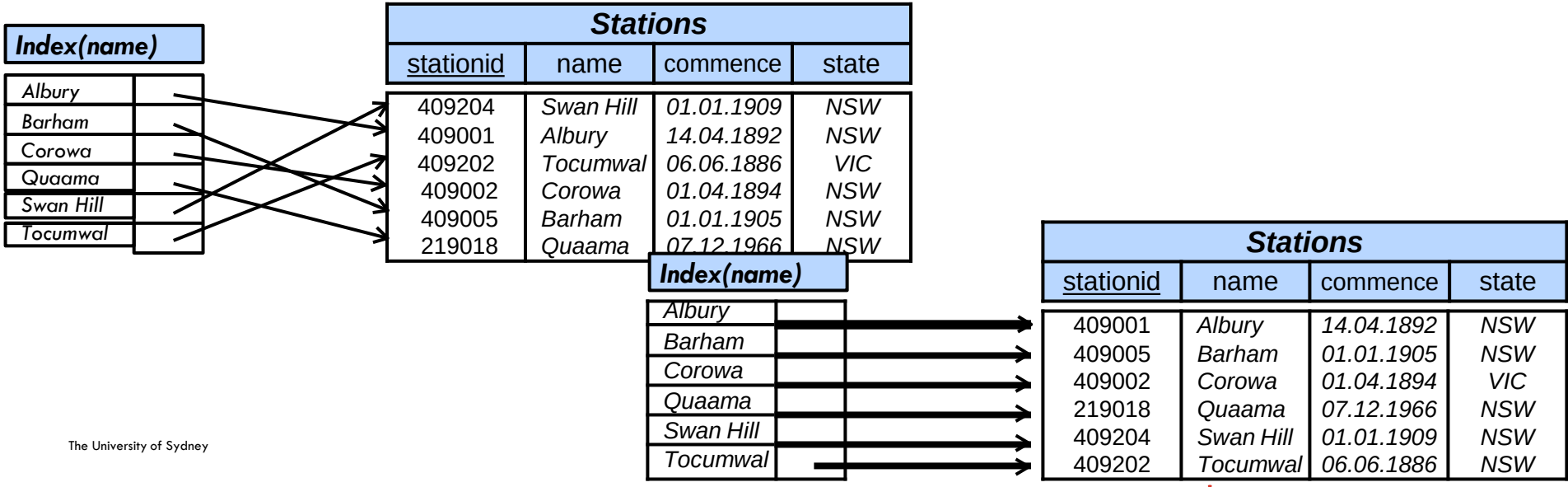
https://www.scaler.com/topics/sql/advantages-and-disadvantages-of-indexing-in-sql/

Summary so far… (Read)

- Indexes are used to speed-up query process in a relational database, resulting in high performance.
- They are similar to textbook indexes.
  - In textbooks, if you need to go to a particular chapter, you go to the index, find the page number of the chapter and go directly to that page.
  - Without indexes, the process of finding your desired chapter would have been very slow.
- Without indexes, a DBMS has to go through all the records in the table in order to retrieve the desired results.
  - This process is called table-scanning and is extremely slow.
- There are two types of Indexes
  1. Clustered Index
  2. Non-Clustered Index

Clustered Index

- A clustered index is an index which defines the physical order in which table records are stored in a database.
  - Both index entries and rows with the actual data are ordered in the same way



Clustered Index contd..

- Since there can be only one way in which records are physically stored in a database table, there can be only one clustered index per table.
- By default a clustered index is created on a primary key column.
- CREATE TABLE statement generally creates a clustered index on primary key.
  - To have clustered index on other attributes :

CLUSTER <Table name> USING <Index Name>

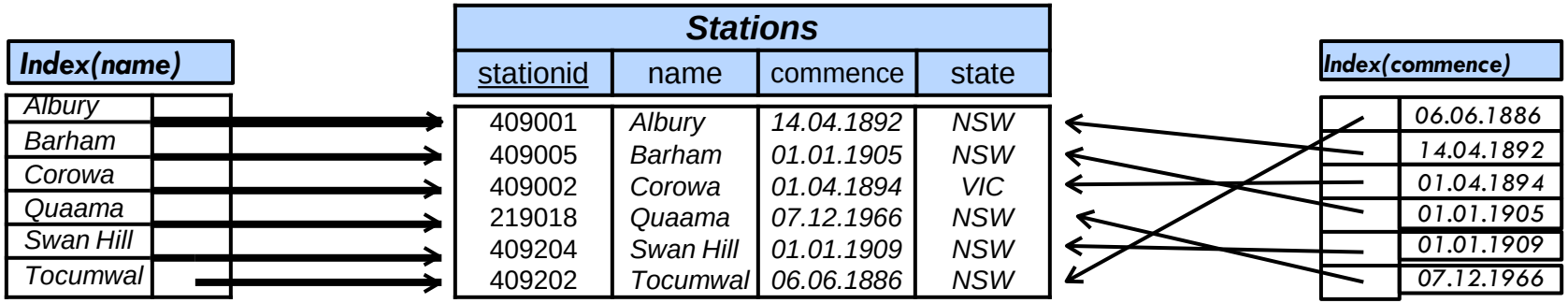
CLUSTER stations USING Stationnameldx

Unclustered Index

- Index entries and rows are not ordered in the same way.
- There can be many secondary indices on a table.
- Index created by CREATE INDEX is generally an unclustered, secondary index.

Unclustered Index Example

- Clustered Index on **name** field of **Stations**
- Unclustered Index on **commence** field of **Stations**



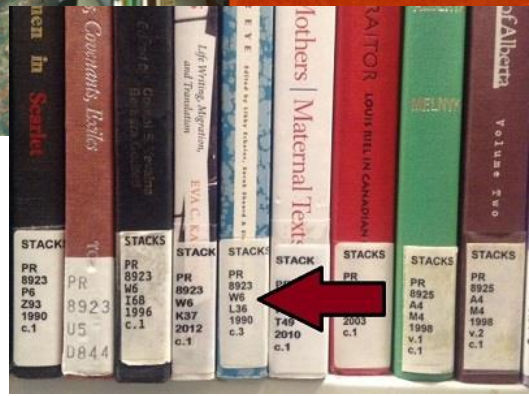
(yes, you can have more than one index on the same table; but only one of those indexes can be clustered)

Comparison

- Clustered index: index entries and rows are ordered in the same way
- There can be at most one clustered index on a table
  - CREATE TABLE generally creates an integrated, clustered (main) index on primary key
- Especially good for "range searches" (where search key is between two limits)
  - Use index to get to the first data row within the search range.
  - Subsequent matching data rows are stored in adjacent locations (many on each block)
  - This minimizes page transfers and maximizes likelihood of cache hits

- Unclustered (secondary) index: index entries and rows are not ordered in the same way
- There can be many unclustered indices on a table
  - As well as perhaps one clustered index
  - Index created by CREATE INDEX is generally an unclustered, secondary index
- Unclustered isn't ever as good as clustered, but may be necessary for attributes other than the primary key

Indexing mimicking the "Physical World"



- Library:

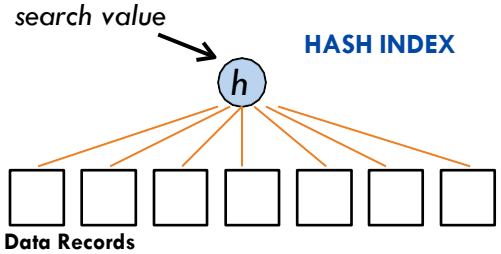
```
CREATE TABLE Library (
        callno      CHAR(10) PRIMARY KEY,
        title       TEXT,
        author      TEXT,
        subject TEXT
 );
```

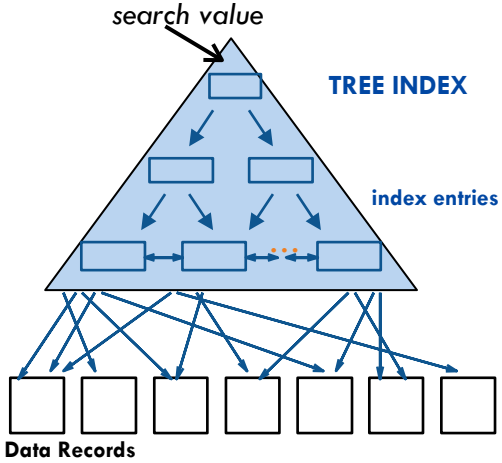
- Library stacks are "clustered" by call number.
- However, we typically search by title, author, keyword
  - The catalog is a **secondary index**…say by Title
- **CREATE INDEX** TitleCatalog **ON** Library(title)

Which Types of Indexes are available?

- Tree-based Indexes: B+-Tree
  - *Very flexible, support point queries, range queries and prefix searches*
- Hash-based Indexes
  - *Fast for equality searches – but nothing else*
- Special Indexes
  - Such as Bitmap Indexes for OLAP or R-Tree for spatial databases

Found in every database engine





Covered in more detail in DATA3404 "Data Science Platforms"

Covering Index

- Goal: Is it possible to answer whole query just from an index?
- **Covering Index** - that provides all the data required for a query without having to access the actual table.
  - When a query is executed, the database looks for the required data in the index tree, retrieves it, and returns the result.
  - This eliminates the need for the engine to access the actual table, saving a secondary traversal to gather the rest of the data.
- Typically a multi-attribute index
- Order of attributes is important: Prefix of the search key must be the attributes from the WHERE

Covering Index Example

Suppose we frequently query the stations table to retrieve the ServiceStationName, Address, postcode and Suburb column

```
SELECT ServiceStationName, Address, Suburb
FROM stations
WHERE Postcode > 2500
```

Create an index having all 4 attributes

```
Create index coveringIdx
on
stations (ServiceStationName, Address, Suburb,Postcode)
```

Platform variation

- Different DBMS support different index structures
  - Tree and/or hash index
  - Special index structures eg for spatial data, or for text data
  - Possibility to *integrate* data records into a clustering index
- And some use unusual terminology for some concepts
  - Eg PostgreSQL says “primary index” for index whose columns incude the primary key

How to Choose Which Indexes to Create?

Understanding the Workload

- For each query in the workload:
  - Which data sets (relations) does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?

- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Choices of Indexes

- One approach: Consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
  - For now, we discuss simple 1-table queries.

- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.

Index Selection Guidelines

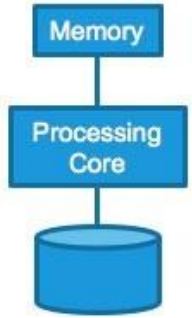
- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests hash index (if the platform provides this option).
  - Range query is only supported by tree index types.
    - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes is important for range queries.
  - Such indexes can sometimes enable **index-only** strategies for important queries (so-called *covering index*).
    - For index-only strategies, clustering is not important!
- Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

Distributed Data Management

Distributed Data Processing

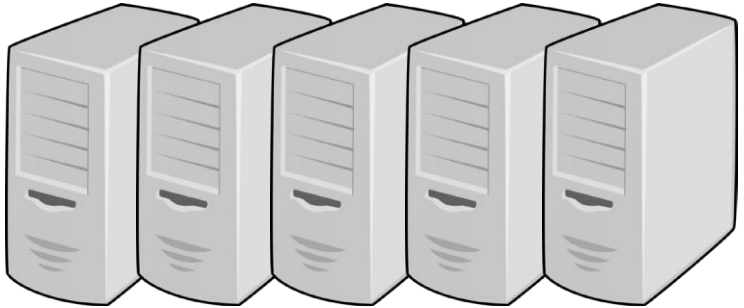
So far: considered **stand-alone** server

Architecture: stand alone



A single server has limits…

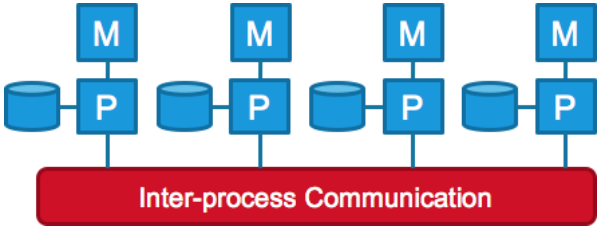
For real Big Data processing, need to **scale-out** to a cluster of multiple servers (nodes):



[Source: Server.png from PinClipart.com]

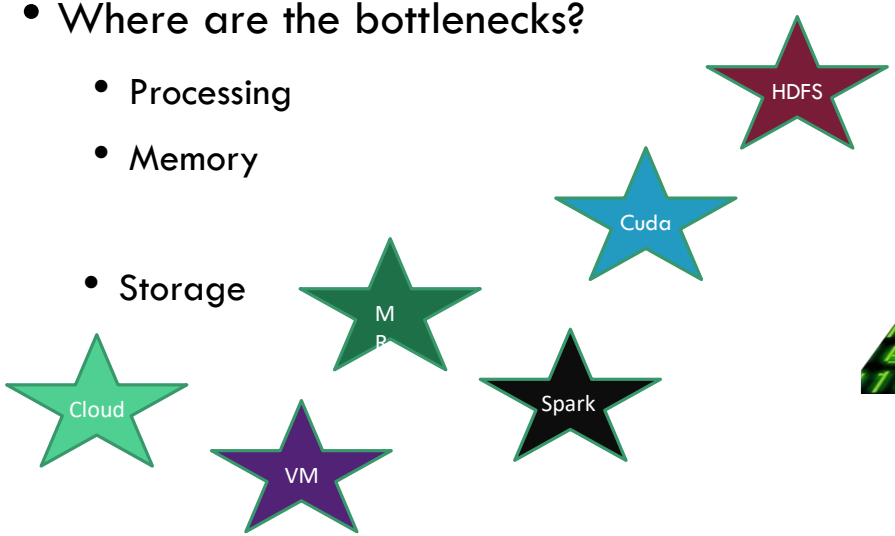
State-of-the-Art:

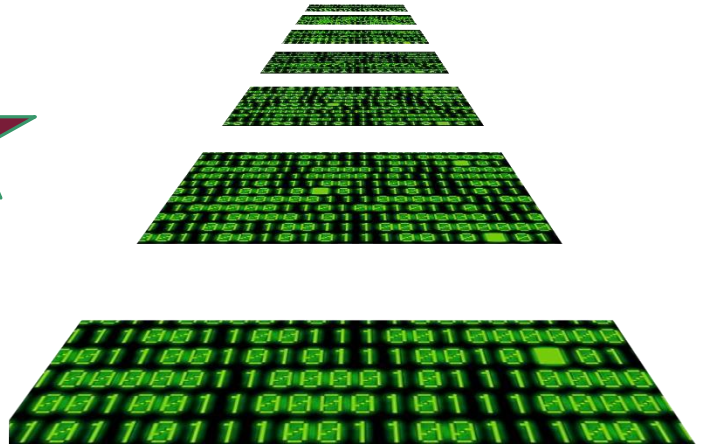
shared-nothing architecture



BIG DATA AND LATENCY

- Having a lot of data means it takes time to process it all
- Where are the bottlenecks?
  - Processing
  - Memory
  - Storage

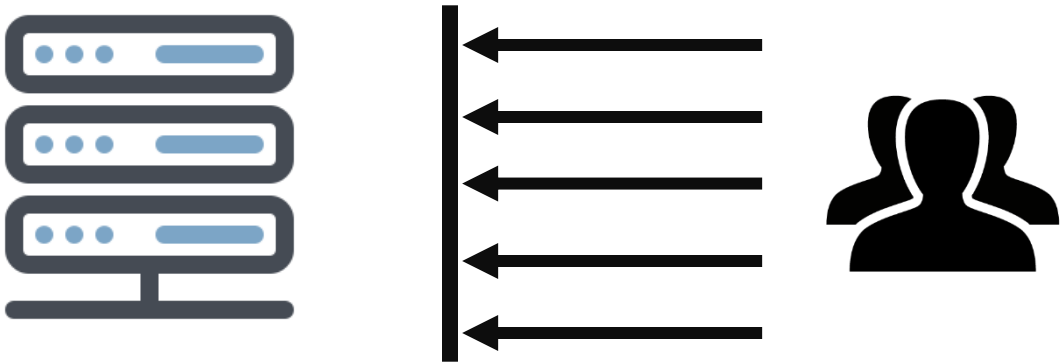




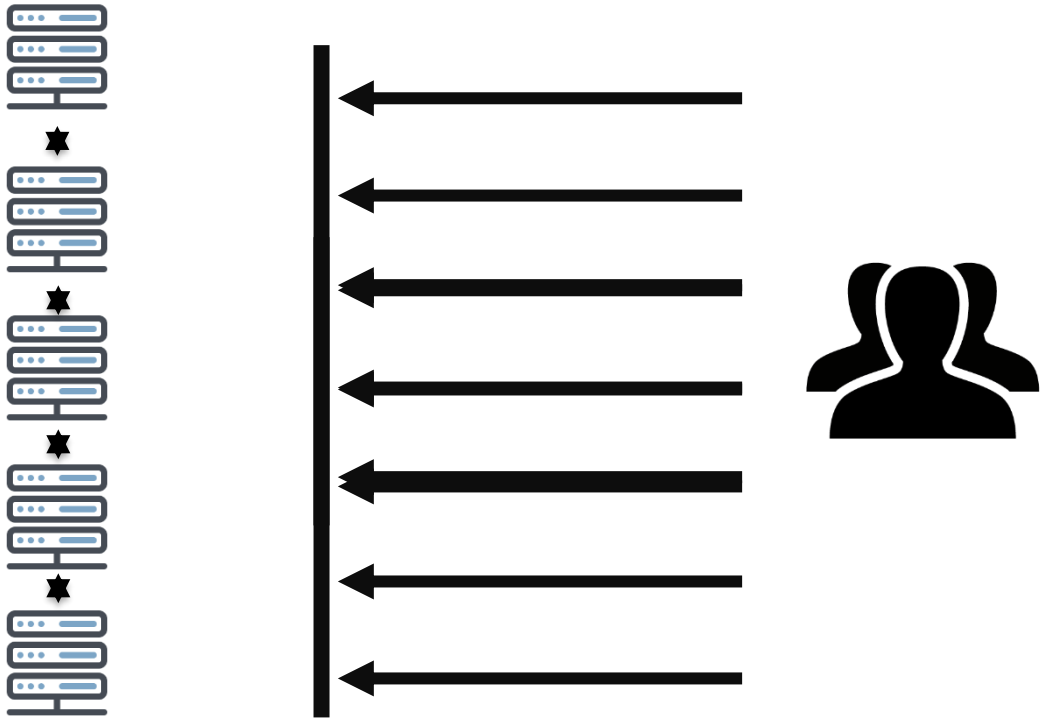
Distributed Computing

A distributed computer system is a computer system that is distributed

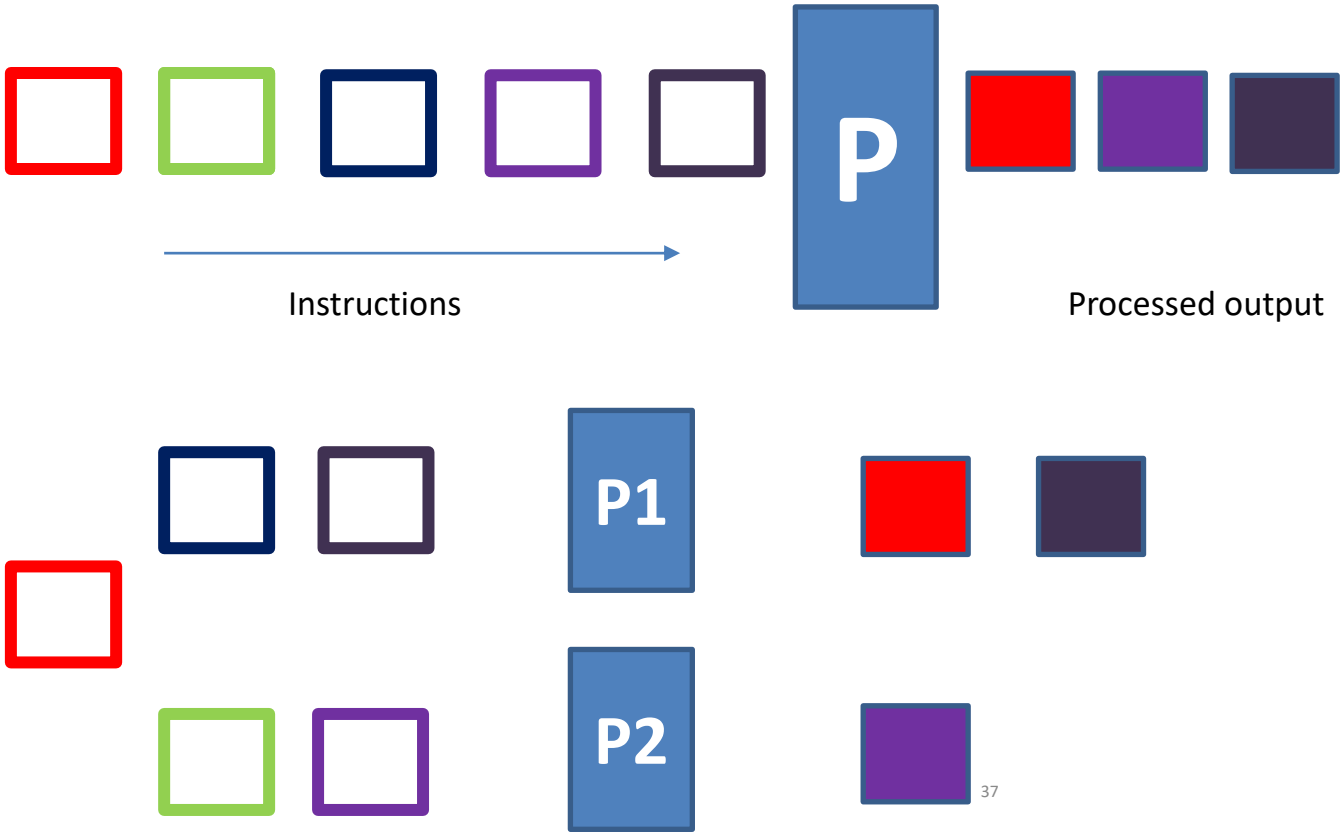
Google receives over approx. 99,000 searches per second



Distributed Computing

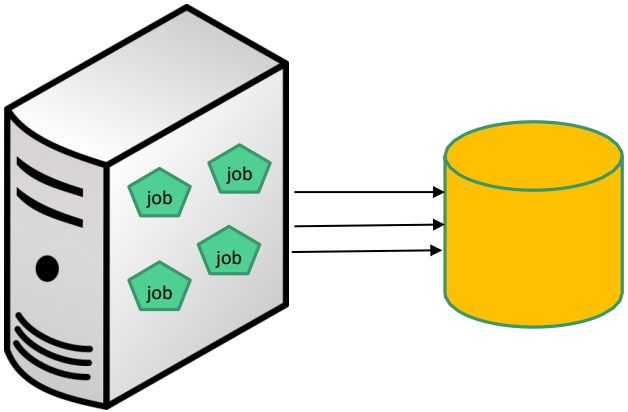


Parallel Computing

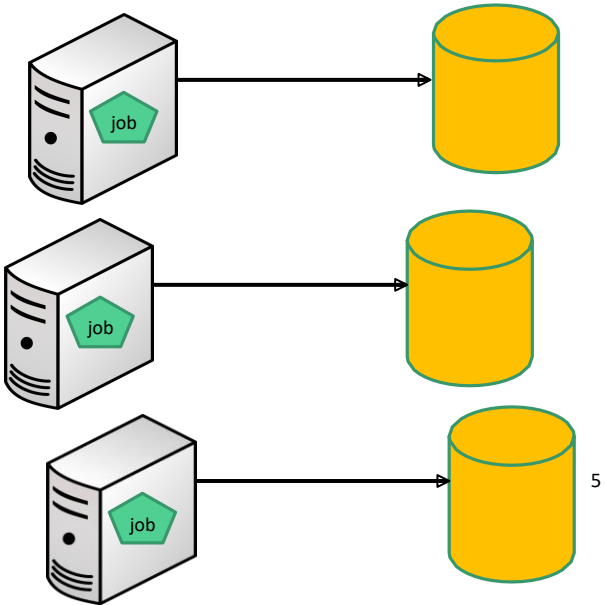


Parallel and distributed processing

- Parallel

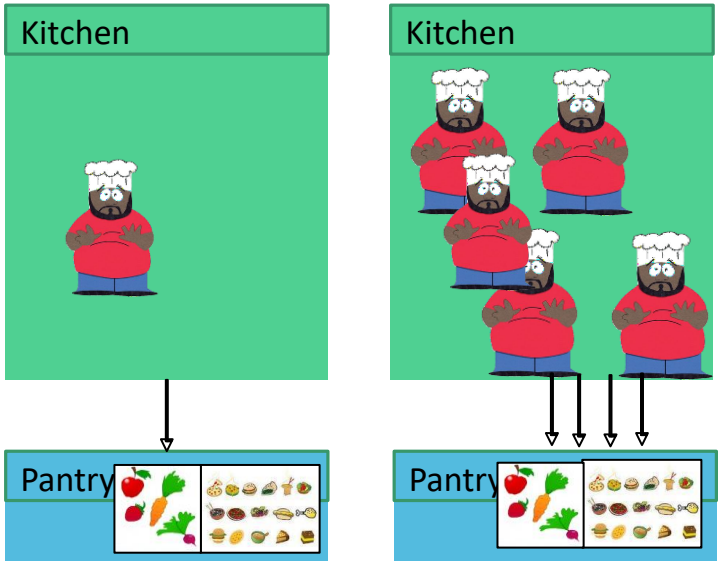


- Parallel and distributed



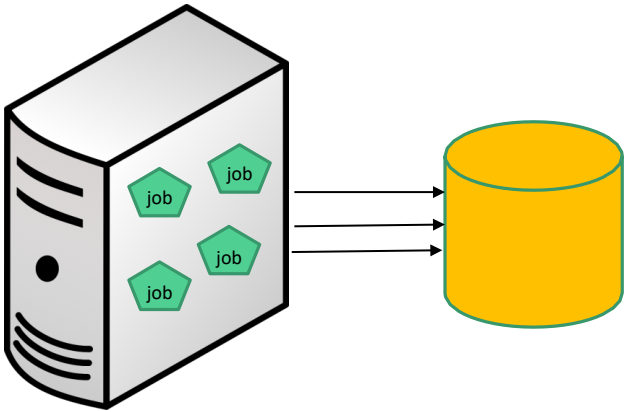
Parallelisation and distribution

- To speed things up, first we parallelise.
  - If your restaurant is busy, a single chef won't be able to keep up.
  - If you need to hire very many chefs, the kitchen will be too small. You have to give every chef their own kitchen.

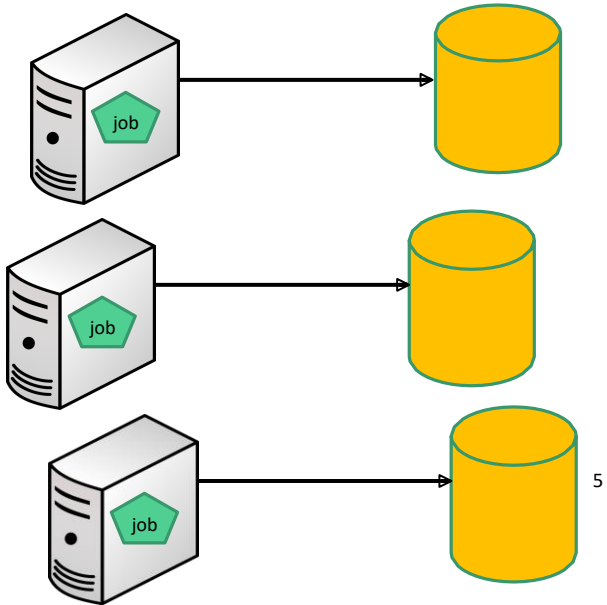


Parallel and distributed processing

- Parallel



- Parallel and distributed



Distributed Data Management

- Two main physical design techniques:
- **Data Partitioning**
  - Storing sub-sets of the original data set at different places
    - can be in different tables in schema on same server, or at remote sites
  - Goal is to query smaller data sets & to gain scalability by parallelism
  - Sub-sets can be defined by
    - columns: ***Vertical Partitioning***
    - rows: ***Horizontal Partitioning***

      (if each partition is stored on a different site also called ***Sharding***)

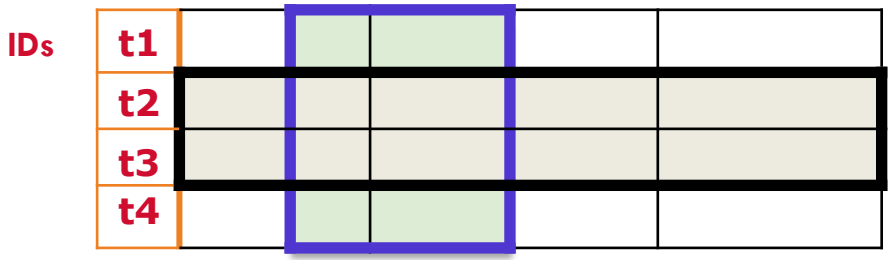
- **Data Replication** (Not covered in this unit of study)
  - Storing copies ('replicas') of the same data at more than one place
  - Goal is fail safety / availability

Data Partitioning

– **What** is partitioned?

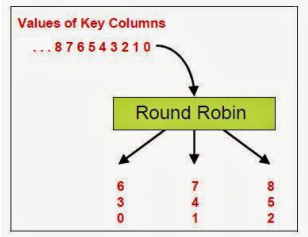
  – **Horizontal Partitioning:** set of rows

  – **Vertical Partitioning:** set of columns



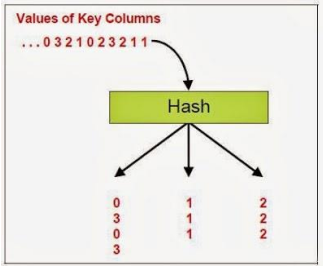
How to place data into partitions?

- **round robin**
  - Placement of partitions is going through all nodes in rounds

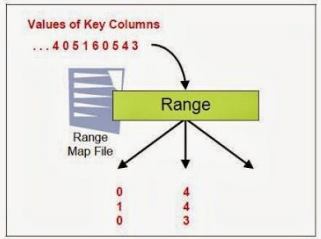


- **hash partitioning**
  - Target node is determined by a hash function on the tuple id or key

https://www.youtube.com/watch?v=KyUTuwz_b7Q&ab_channel=ComputerScience



- **range partitioning**
  - Each node stores a partitioned defined by a range predicate



Example

Imagine we have this *Stations* table.
How would we horizontal/vertical partition into two tables?

Stations

stationid	name	commence	state
409001	Murray River at Albury	14.04.1892	NSW
409002	Murray River at Corowa	01.04.1894	NSW
409003	Murray River at Denuquin	01.09.1899	NSW
409004	Murray River at Barham	01.01.1905	NSW
409202	Murray River at Tocumwal	06.06.1886	VIC
409204	Murray River at Swan Hill	01.01.1909	VIC
219018	Murray River at Quaama	07.12.1966	VIC

Horizontal Partitioning of Stations table by *state*

stationid	name	commence	state
409001	Murray River at Albury	14.04.1892	NSW
409002	Murray River at Corowa	01.04.1894	NSW
409003	Murray River at Denuquin	01.09.1899	NSW
409004	Murray River at Barham	01.01.1905	NSW

$partition_1$ = *Stations* where state='NSW'

stationid	name	commence	state
409202	Murray River at Tocumwal	06.06.1886	VIC
409204	Murray River at Swan Hill	01.01.1909	VIC
219018	Murray River at Quaama	07.12.1966	VIC

$partition_2$ = *Stations* where state='VIC'

Q: Which kind of horizontal partitioning is this?

Vertical Partitioning of Stations table

stationid	commence	state
409001	14.04.1892	NSW
409002	01.04.1894	NSW
409003	01.09.1899	NSW
409004	01.01.1905	NSW
409202	06.06.1886	VIC
409204	01.01.1909	VIC
219018	07.12.1966	VIC

$partition_1$ = stationid, commence, state columns from Stations

stationid	name
409001	Murray River at Albury
409002	Murray River at Corowa
409003	Murray River at Denuquin
409004	Murray River at Barham
409202	Murray River at Tocumwal
409204	Murray River at Swan Hill
219018	Murray River at Quaama

$partition_2$ = stationid, name columns from Stations

Note: primary key column is in both vertical partitions

Partitioning and Data Sharding

- **Advantages of Partitioning:**
  - **Improve performance** (reduce computational time)
  - **Scalability** (more partition can be added)
    if one partition is down, others are unaffected **if** stored on different tablespace / disk
  - **Load balancing** (workload can be distributed more evenly)
  - **Fault tolerance** (data can be replicated across partition)

- **Data Sharding (Fragmenting)**: Distributing data partitions over several sites in a distributed database
  - Assumes queries only access one shard only
  - Otherwise, counter-productive (eg. if we need to join two partitioned tables which are no *co-located* per site…
    - Co-located: join tuples are on the same node, e.g. by partitioning both tables on the join attr.

Conclusions

Lessons Learned this Week

- **Physical Data Organisation**
  - Core Problem: Persistency requires disks, but those are SLOW
- **Understanding of the concept of an Index**
  - Efficient access to single tuples or even ranges based on *search keys*
  - One *primary*, but several *secondary indices* possible per relation
- single- vs. *multi-attribute indices*, *clustered* vs. *unclustered indices*
- **Practical experience with indexing a relational database**
  - How to suggest appropriate indexes for a given SQL workload
  - Awareness of the trade-off between SQL query performance and indexing costs (updates)
- **Distributed Data Management**
  - Data Partitioning; Data Replication

References

- Kifer/Bernstein/Lewis (2nd edition)
  - Chapter 9 (9.1-9.4)
  - Chapter 12 (database tuning)
  - *Kifer/Bernstein/Lewis gives a good overview of indexing and especially on how to usethem for database tuning. Thisis the focus for isys2120 too.*
- Ramakrishnan/Gehrke (3rd edition - the 'Cow' book)
  - Chapter 8
  - *TheRamakrishnan/Gehrke is very technical on this topic, providing a lot of insight into how disk-based indexes are implemented. We only need the overview here (Chap8); technical details are covered in data3404.*
- Ullman/Widom (3rd edition - '1st Course in Databases')
  - Chapter 8 (8.3 onwards)
  - *Mostly overview, but cost model of indexing goes further than we discuss here in the lecture*
- Abadi, Boncz, Harizopoulos, Idreos, Madden:
  "The Design and Implementation of Modern Column-Oriented Database Systems",
  Foundations and Trends in Databases, Vol 5, No 3, 2012.
- [Oracle 10g Database Concepts, Chap. 5.4]
  Oracle Corporation: Oracle 10g Documentation, *Database Concepts.*