SOEN 363 - Summary

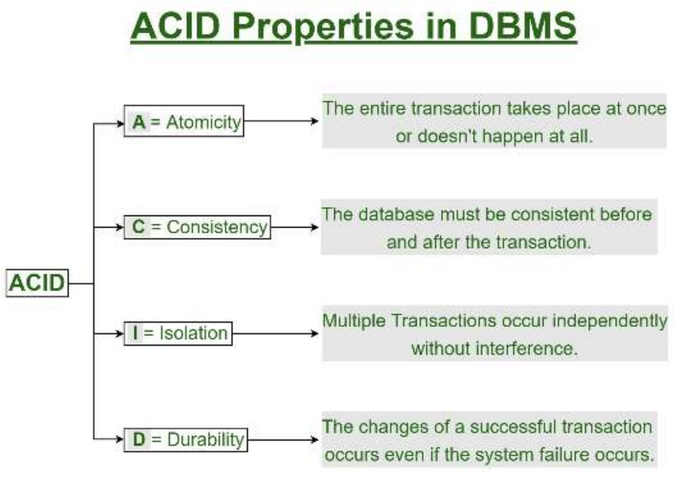
Data Systems for Software Engineers (Concordia University)

Introduction

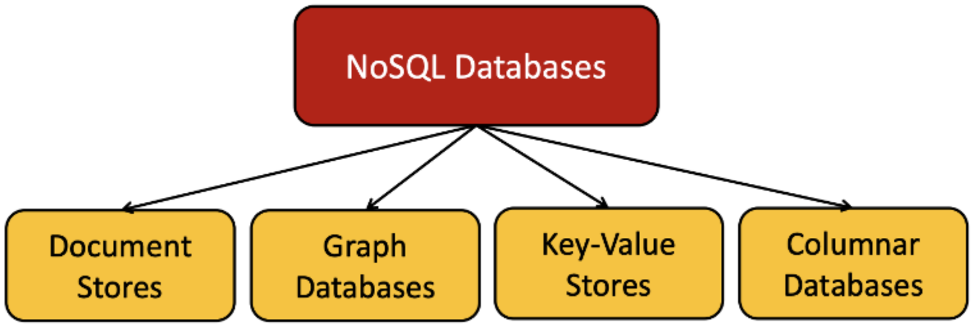
Data exploding along three main dimensions:
- Volume
- Velocity
- Variety

Big Data: Proliferation of data that floods organizations on a daily basis. High velocity/volume/variety information assets.

Database Management Systems (DBMS): Needed to record, maintain, access and manipulate data correctly, securely, efficiently and effectively..


ACID Properties in DBMS
ACID
A = Atomicity
The entire transaction takes place at once or doesn't happen at all.
C = Consistency
The database must be consistent before and after the transaction.
I = Isolation
Multiple Transactions occur independently without interference.
D = Durability
The changes of a successful transaction occurs even if the system failure occurs.

NoSQL Databases
- No strict schema requirements
- No strict adherence to ACID properties
- Consistency is traded in favor of availability



Databases Introduction

Database: Collection of data which describes one or many real-world enterprises

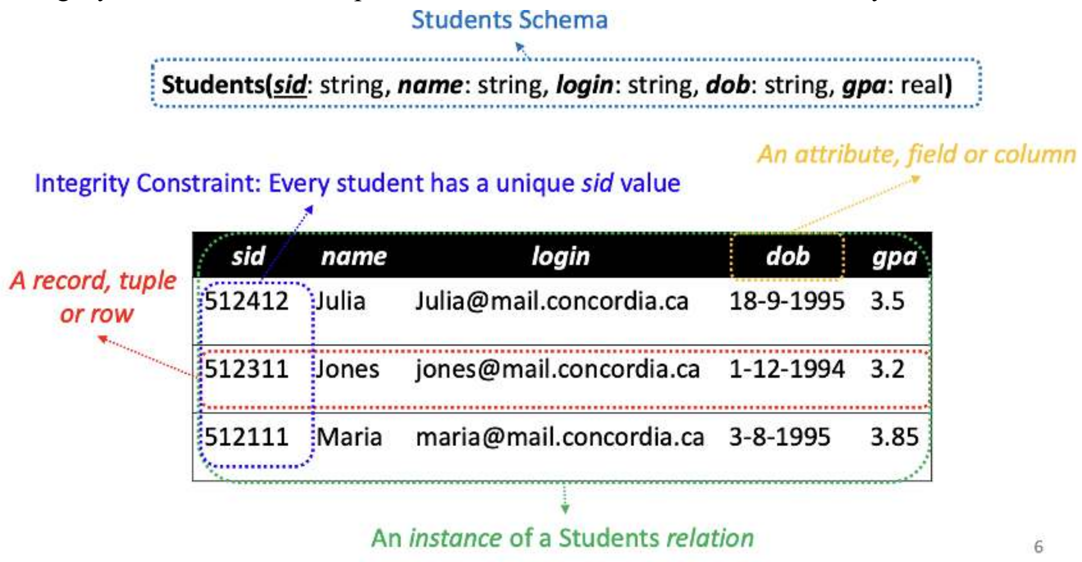
DBMS: Software package designed to store and manage databases.

Database System: (Big) Data + DBMS + Application Programs

Data Model: Collection of high-level data description constructs that hide many low-level storage details.

Relational Model

- One of the most widely used models
- Central data description in the model is the *relation*
- Integrity Constraints can be specified: Conditions that records must satisfy

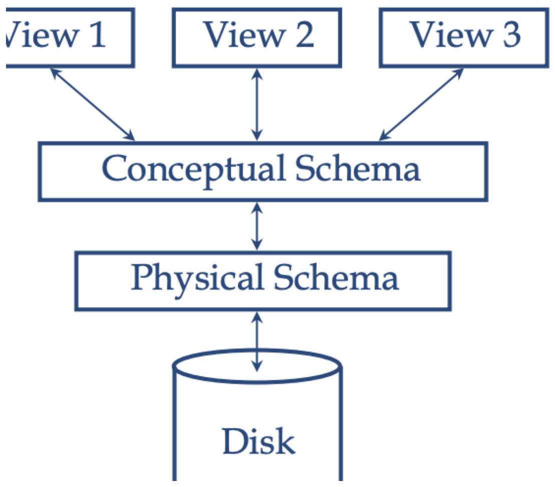

Students Schema
Students(sid: string, name: string, login: string, dob: string, gpa: real)
An attribute, field or column
Integrity Constraint: Every student has a unique sid value
A record, tuple or row
An instance of a Students relation
6

sid	name	login	dob	gpa
512412	Julia	Julia@mail.concordia.ca	18-9-1995	3.5
512311	Jones	jones@mail.concordia.ca	1-12-1994	3.2
512111	Maria	maria@mail.concordia.ca	3-8-1995	3.85

Levels of Abstraction

Conceptual (or logical), physical and external schemas

- External (or Views): Allow data access to be customized at the level of users or group of users..
- Conceptual: Describes Data in terms of a specific data model (e.g., the relation model of data)
- Physical: Specifies how data described in the conceptual schema are stored on secondary storage devices.



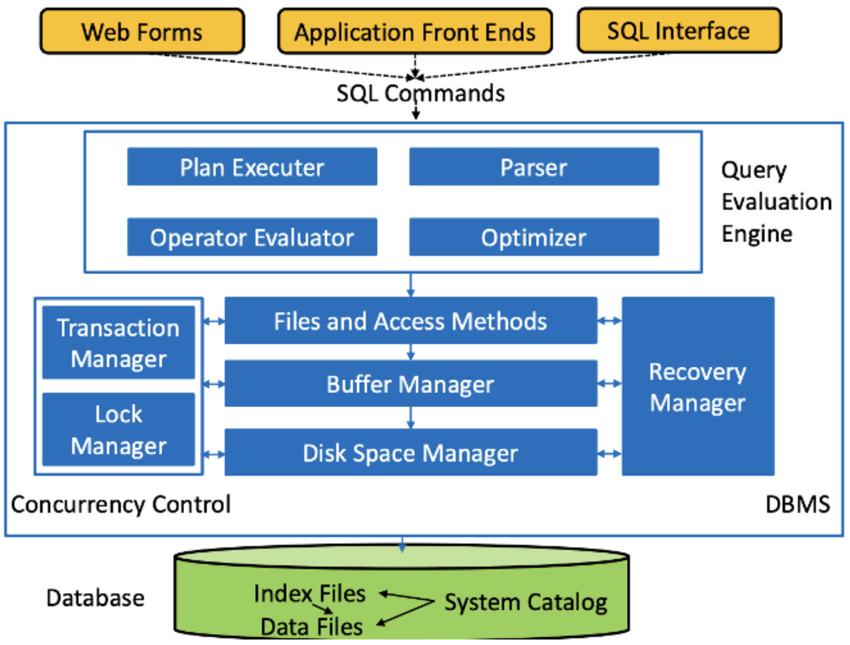
Data Independence
One of the most important benefits of using a DBMS

- Applications programs are insulated from how data is structured and stored

2 Properties

1. Logical data Independence: Users are shielded from changed in the conceptual schema (eg: add/drop column in a table)
2. Physical Data Independence: Users are shielded from changed in the physical schema (eg add index or change record order)

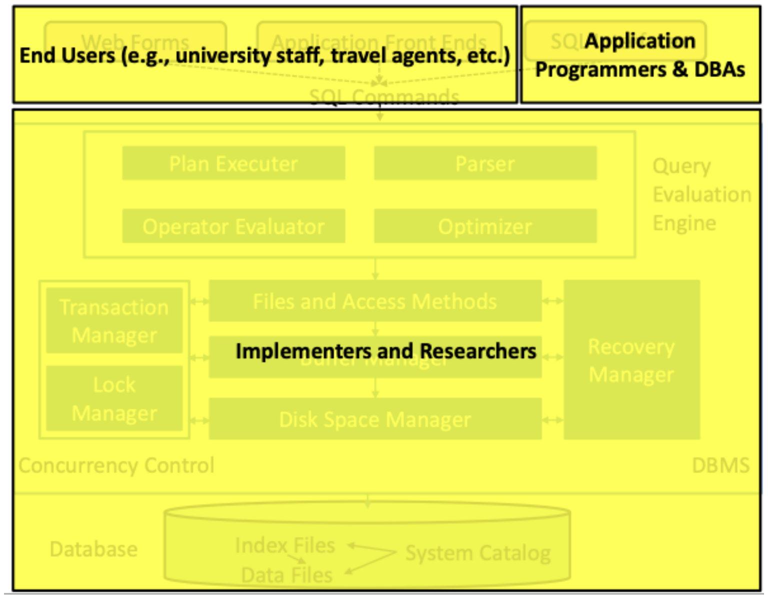
DBMS



- DBMS Need to schedule concurrent accesses to data to improve performance
- Must order requests carefully to avoid conflicts
    - Use *locking protocol* to ensure conflicts do not arise

Atomicity : All-or-Nothing property, even if a crash occurs in the middle of the transaction.

- Achieved via maintaining a log of all writes to the database
    - Before a change is made to the db, corresponding log entry is forced to a safe location (Write-Ahead Log)
    - After a crash, the effects of partially executed transactions are undone using the log



1. End Users
2. Application Programmers
3. Database Administrators (DBAs)
4. Implementers
5. Researchers

Database Design

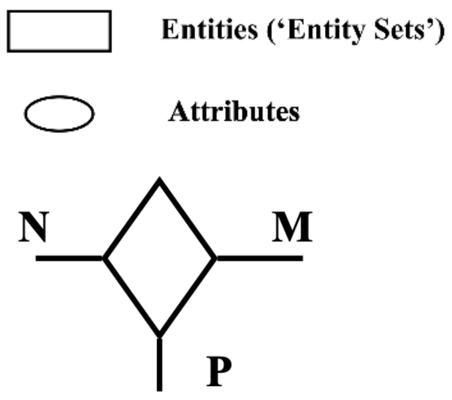
- Requirement Analysis
    - User needs
- Conceptual Design
    - High-level description of the data
- Logical Design
    - Conversion of an ER design into a relational database schema
- Schema Refinement
    - Normalization (i.e restructuring tables to ensure some desirable properties)
- Physical Design
    - Building indexes and clustering some tables
- Security Design
    - Access Control

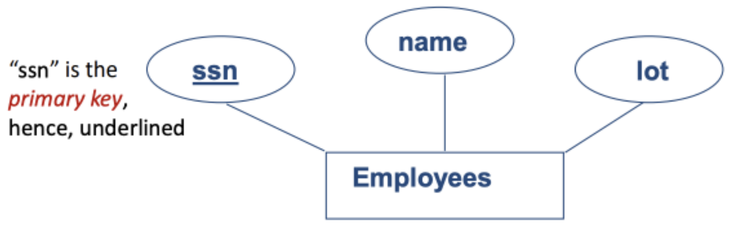
ER Module

Entity
- Real-World object distinguishable from other objects in an enterprise
- Described through a set of attributes

Entity Set
- Collection of similar entities (eg. all students)
- All entities in the set have the same set of attributes
- Each entity has a *key*
- Each attribute has a *domain*

ER Diagram

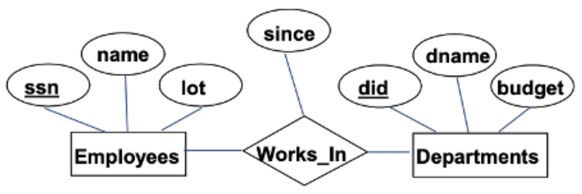




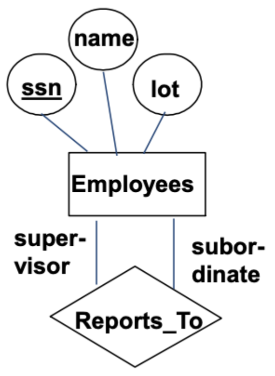
▪ Nouns -> entity sets
▪ Verbs -> relationship sets

Relationship
- Association among two or more entities
- Described using a set of attributes

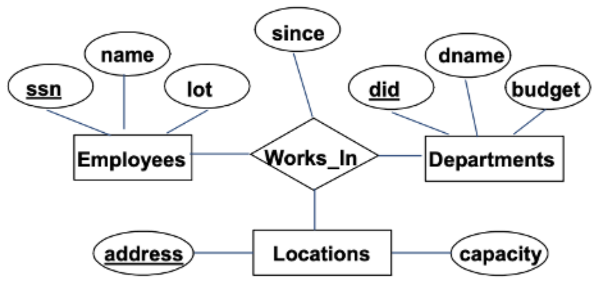
Relationship set
- Collection of similar relationship
- Same entity set could participate in different relationship sets, or different "roles" in the same set



A Binary Relationship



A Self-Relationship



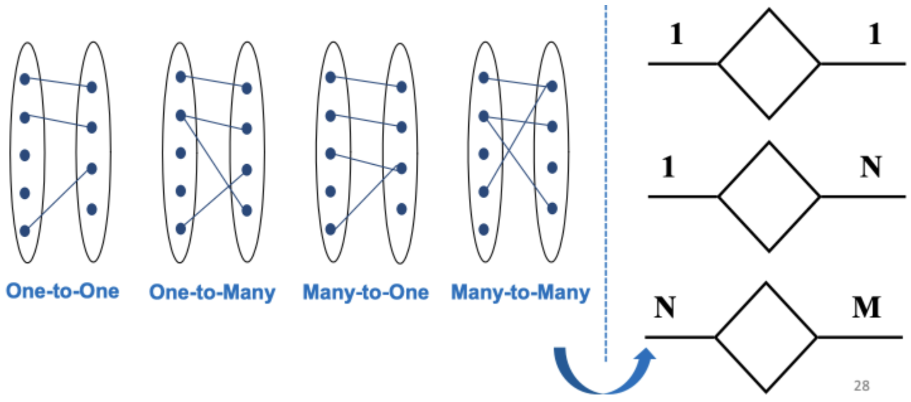
Ternary Relationship

Key Constraints

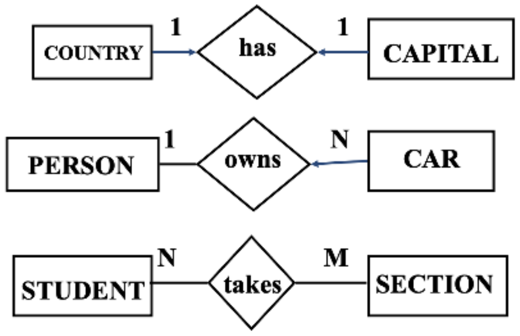
Eg.

- An employee can work in many departments
- A department can have many employees

Cardinalities

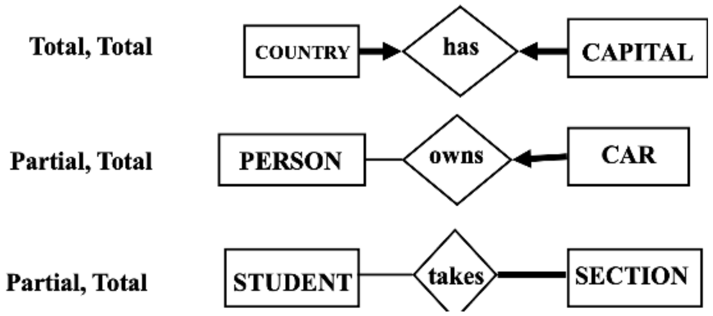


Eg.



Total vs Partial Participation

Total: Thick Line

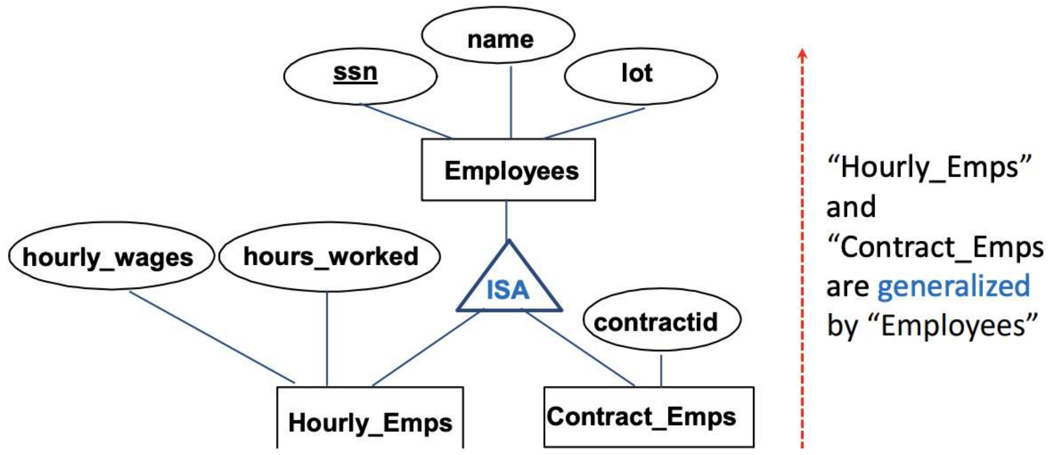


Weak Entity: Can be identified uniquely by considering the primary key of another owner entity.

- Its set of attributes that uniquely identifies a weak entity for a given owner entity is called *partial key*.

ISA hierarchy

- Like OOP
- Attributes are inherited
- We can have many levels of ISA hierarchy
- Reasons to use it:
    - Add descriptive attributes to subclasses
    - Identify entities that participate in a relationship
- Eg. If we declare B ISA A, every B entity is also considered an A entity.



- Overlap Constraint: Can an entity belong to B <u>and</u> C
- Covering Constraint: Can an A entity belong to <u>neither</u> B or C

Conceptual Design

Entity vs. Attribute

Eg. Address should be an attribute of Employee or an entity connected to employee by a relationship

- If we have several addressed per employee: Address must be an entity
- If the structure (city, street etc) is important, address must be an entity

Relational Model

Adopts a tabular representation

- Database is a collection of one or more relations
- Each relation is a table with rows and columns

Unique because of its simple data representation and ease with which complex queries can be expressed

Relationship

1. A *schema* which includes:
    a. The relation's name
    b. The name of each column
    c. The domain of each column (Specifies a condition by which each instance of a relation should satisfy)
        i. Defined by a DBA
        ii. Enforces by the DBMS
2. An *instance* (Set of tuples)
    a. Each tuple has the same number of columns as the relation schema

▪ What is the relational database schema (not the relation schema)?

- A collection of schemas for the relations in the database

▪ What is the instance of a relational database (not the instance of a relation)?

- A collection of relation instances

Basic SQL

SQL (Sequel) => Structured Query Language

DDL: Data Definition Language

- Creating, modifying and deleting relations and views
- Allows specifying constraints
- Administering users, security etc

DML: Data Manipulation Language

- Posing queries to find tuples that satisfy criteria
- Adding, modifying and removing tuples

Note: DBMS enforces domain constraints whenever tuples are added or modified

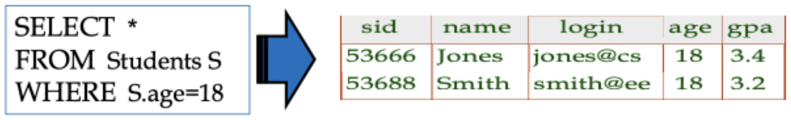
```
INSERT INTO  Students (sid, name, login, age, gpa)
VALUES  (53688, 'Smith', 'smith@ee', 18, 3.2)
```

```
DELETE
FROM Students S
WHERE S.name = 'Smith'
```

Querying a relation:

How can we find all 18-year old students?

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

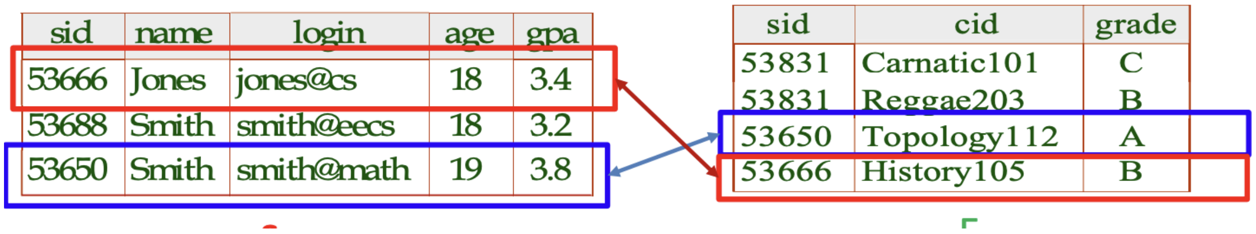


```
SELECT  *
FROM  Students S
WHERE  S.age=18
```

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2

Querying multiple relations:

```
SELECT  S.name, E.cid
FROM  Students S, Enrolled E
WHERE  S.sid=E.sid AND E.grade="A"
```



sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

sid	cid	grade
53831	Carnatic101	C
53831	Reggae203	B
53650	Topology112	A
53666	History105	B

```
ALTER TABLE  Students
     ADD COLUMN firstYear: integer
```

DROP TABLE table_name; //Destroys the relation

//Add a column

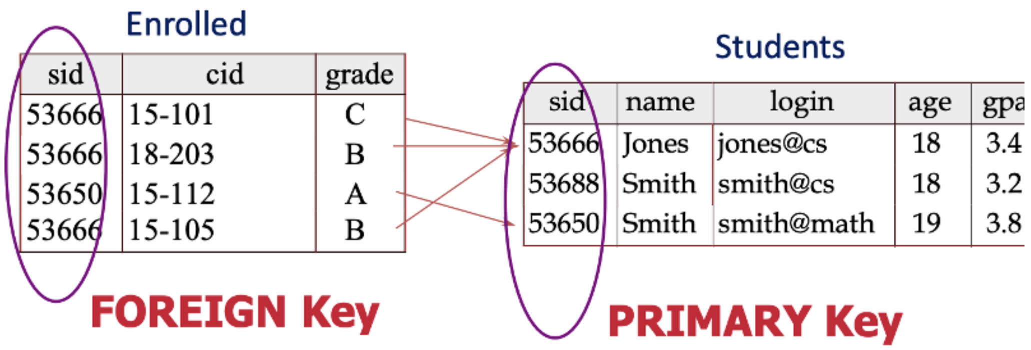
Integrity Constraints (ICs)

- Condition that must be true for any instance of the database
  - Specified when schemas are defined
  - Checked when relations are modified
- Legal Instance: Satisfies all specified ICs

- We can check a database instance to see if an IC isviolated, but we can NEVER infer that an IC is true by looking at an instance
-

Keys
Help associate tuples in different relations

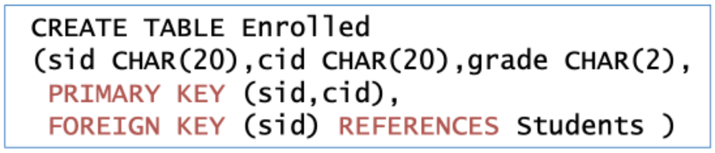
A form of Integrity constraints



sid	cid	grade
53666	15-101	C
53666	18-203	B
53650	15-112	A
53666	15-105	B

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@cs	18	3.2
53650	Smith	smith@math	19	3.8

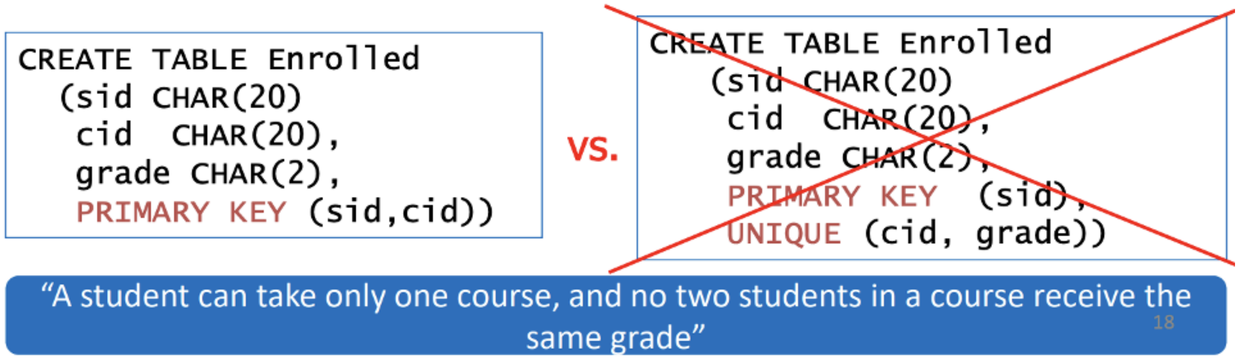
Superkey: No two distinct tuples can have same values in all key fields
Primary key: Minimal superkey
Candidate key: Other keys that can be primary key but aren’t. Specified using UNIQUE
Foreign Key: Set of fields referring to a tuple in another relation (acts like logical pointer)



```
CREATE TABLE Enrolled
(sid CHAR(20),cid CHAR(20),grade CHAR(2),
 PRIMARY KEY (sid,cid),
 FOREIGN KEY (sid) REFERENCES Students )
```

Eg.
sid is a key for Students (what about name?)
The set {sid, name} is a superkey (or a set of fields that contains a key)

Or “For a given student and course, there is a single grade”

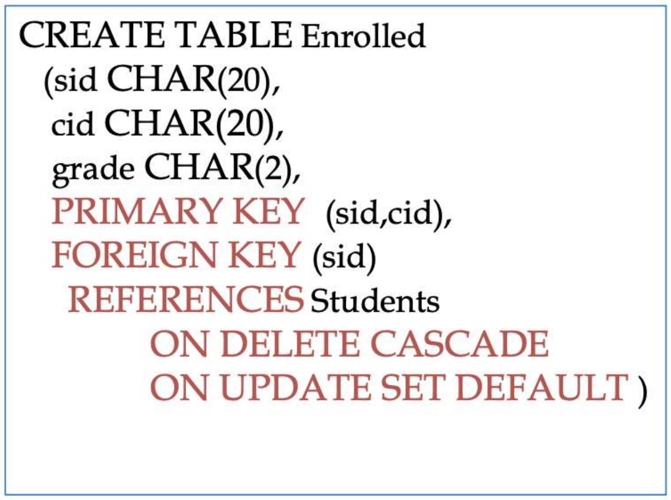


```
CREATE TABLE Enrolled
  (sid CHAR(20)
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid))
```

VS.

~~CREATE TABLE Enrolled~~
~~(sid CHAR(20)~~
~~cid CHAR(20),~~
~~grade CHAR(2),~~
~~PRIMARY KEY (sid),~~
~~UNIQUE (cid, grade))~~

“A student can take only one course, and no two students in a course receive the same grade”

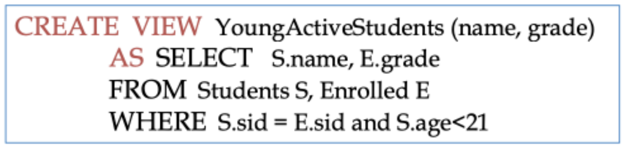


```
CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid,cid),
   FOREIGN KEY (sid)
    REFERENCES Students
          ON DELETE CASCADE
          ON UPDATE SET DEFAULT )
```

Default is NO ACTION (i.e., delete/update is rejected)
▪ CASCADE (also delete all
tuples that refer to the
deleted tuple)
▪ SET NULL / SET DEFAULT (sets foreign key value of referencing tuple)

Views
A table whose rows are not explicitly stored but computed as needed.

- Allows applications to transparently assume the old schema if changed (Logical Data Independence)
- Can be defined to give a group of users access to just the information they are allowed to see (SECURITY)
-



```
CREATE  VIEW  YoungActiveStudents (name, grade)
          AS  SELECT   S.name, E.grade
          FROM  Students S, Enrolled E
          WHERE  S.sid = E.sid and S.age<21
```

DROP VIEW command. (IF we DROP TABLE a table that has a view, we need to specify how to handle it)

SQL Major Aspects

Query Languages != Programming Languages

Not meant for complex calculations

They support easy and efficient access to large datasets

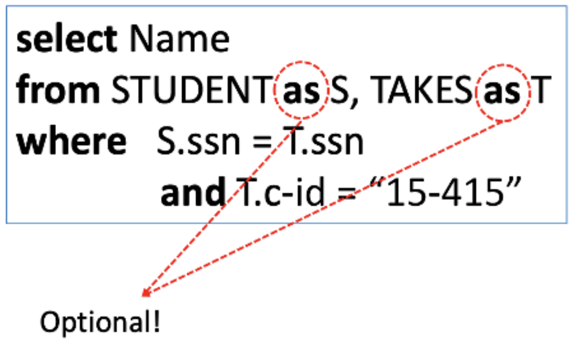
The correct order of execution in SQL is FROM, WHERE, GROUP BY, HAVING, SELECT, DISTINCT, ORDER BY and LIMIT.

Main SQL Aspects:

1. Data Manipulation Language (DML)
    a. Allows users to pose queries and insert/delete/modify rows(tuples)
2. Data Definition Language (DDL)
    a. Allows users to create/delete/modify views and tables
3. Triggers and Advanced Integrity constraints
    a. Actions executed by the DBMS whenever changes to the database meet specific conditions
4. Embedded and Dynamic Language
    a. Embedded: Allow SQL code to be called from a host language
    b. Dynamic: Allow SQL queries to be constructed and executed at run-time
5. Remote Database Access
6. Transaction Management
7. Security
8. And more…

WHERE Clause

- Boolean Operators (and, or, not)
- Comparison Operators (<, ≤, >, ≥, =, ≠)
- % and _ for strings
    - %: Variable-length (i.e., stands for 0 or more arbitrary characters)
    - _: Single-character (i.e., stands for any 1 character)



```
select Name
from STUDENT as S, TAKES as T
where   S.ssn = T.ssn
        and T.c-id = "15-415"
```

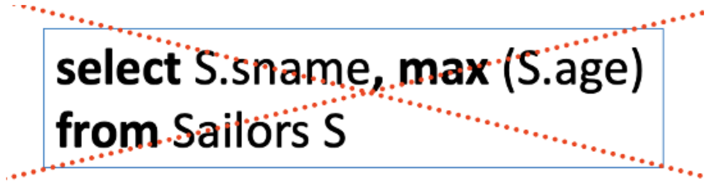
Set Operations

Intersect: Returns distinct rows that are returned by both input queries.
Union: Returns distinct results.
Except: Returns distinct rows from the left input query that are not returned by the right input query.

Aggregate Functions And Clauses

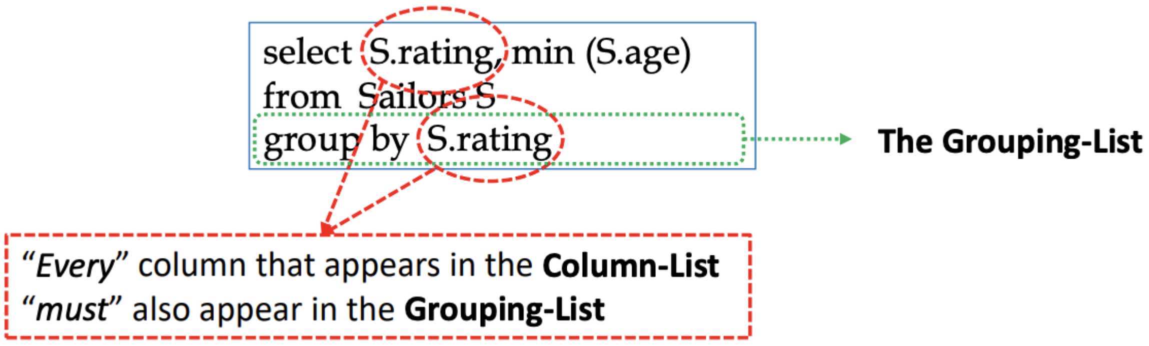
avg(), count(), max() etc

Note: If the "select" clause uses an aggregate function, itmust use ONLY aggregate function unless the query contains a "group by" clause!



```
select S.sname, max (S.age)
from Sailors S
```

HAVING ⇒ Added to SQL because Where cannot be used with aggregate functions
GROUP BY ⇒ Combines similar rows, producing a single result row for each group of rows that have the same values



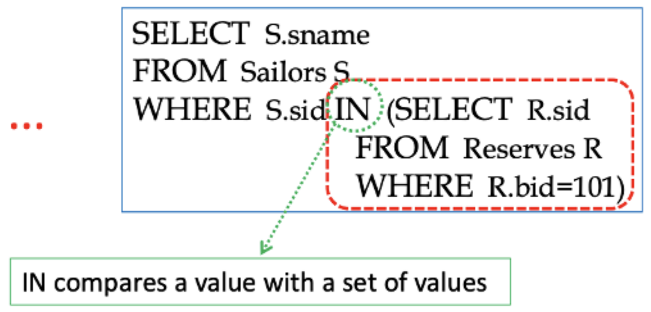
```
select S.rating, min (S.age)
from  Sailors S
group by S.rating
```

The Grouping-List
"Every" column that appears in the Column-List "must" also appear in the Grouping-List

Eg. Lists the number of customers in each country, sorted high to low (Only include countries with more than 5 customers)

```sql
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5
ORDER BY COUNT(CustomerID) DESC;
```

Nested Queries

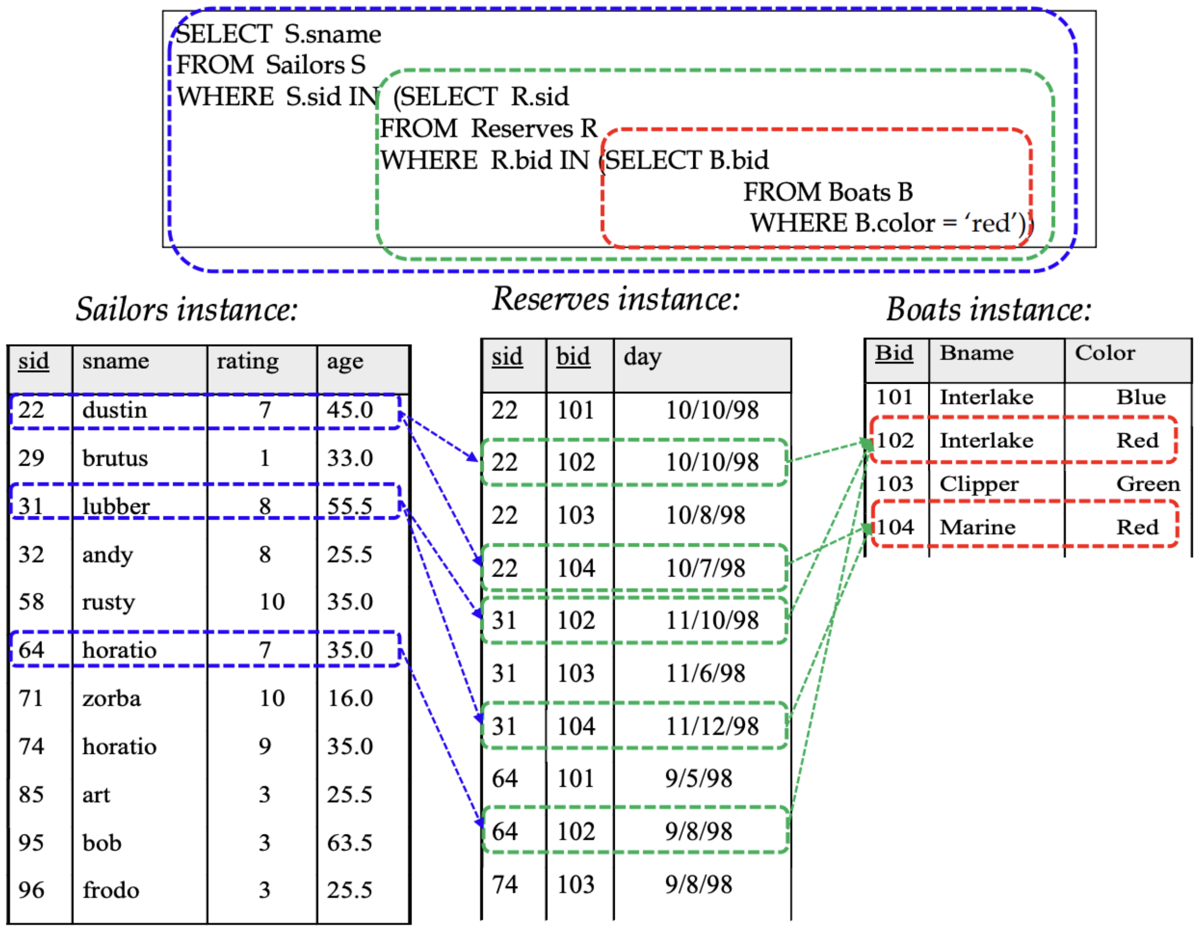


```
SELECT  S.sname
FROM  Sailors S
WHERE  S.sid IN  (SELECT  R.sid
                  FROM  Reserves R
                  WHERE  R.bid=101)
```

==>Find the names of sailors who have reserved boat 101

Use NOT IN, for the names of the sailors who have not reserved boat 101
Having clause can also include subqueries

Deeply Nested Queries:



```
SELECT  S.sname
FROM  Sailors S
WHERE  S.sid IN  (SELECT  R.sid
                  FROM  Reserves R
                  WHERE  R.bid IN (SELECT B.bid
                                          FROM Boats B
                                           WHERE B.color = 'red'))
```

Sailors instance:

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

Reserves instance:

sid	bid	day
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Boats instance:

Bid	Bname	Color
101	Interlake	Blue
102	Interlake	Red
103	Clipper	Green
104	Marine	Red

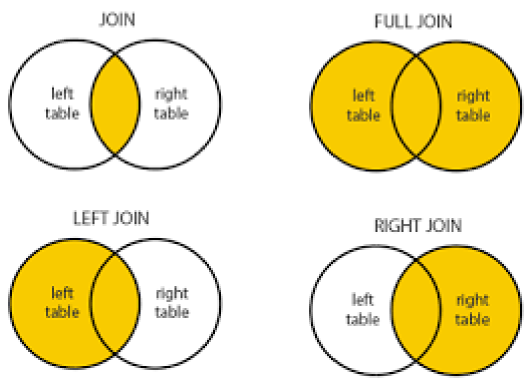
NULL Value

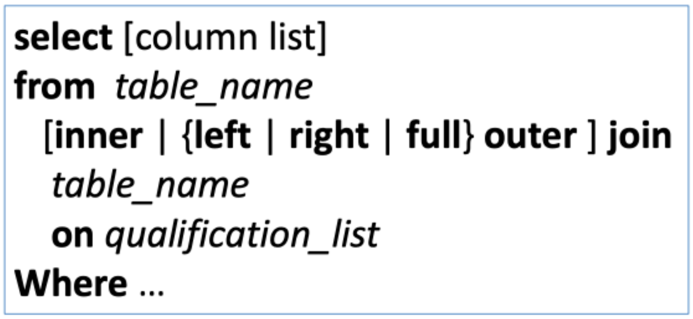
- Count(*) handles null like any other value
- Sum, avg, min and max discard NULL values

Comparison:

- NOT unknown → unknown
- True OR unknown → True
- False OR unknown → unknown
- False AND unknown → False
- True AND unknown → unknown
- Unknown [AND|OR|=] unknown → unknown

Joins

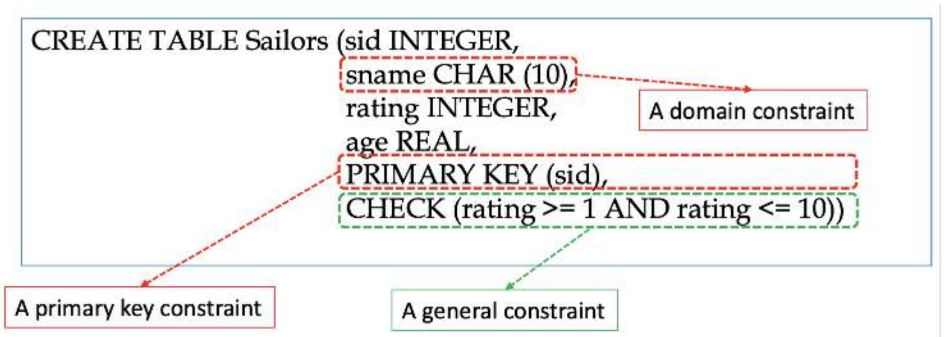




select [column list]
from *table_name*
  [**inner** | {**left** | **right** | **full**} **outer**] **join**
  table_name
  on *qualification_list*
Where …

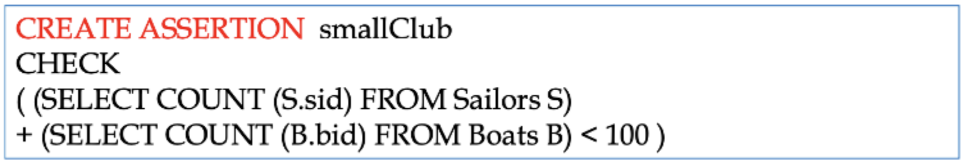
Integrity Constraints (Review)

- Domain Constraints
- Primary Key Constraints
- Foreign Key Constraints
- General Constraints
  - Eg. CHECK



```
CREATE TABLE Sailors (sid INTEGER,
                      sname CHAR (10),
                      rating INTEGER,
                      age REAL,
                      PRIMARY KEY (sid),
                      CHECK (rating >= 1 AND rating <= 10))
```

How can we enforce that the number of boats plus the number of sailors should not exceed 100? *Assertions*



```
CREATE ASSERTION  smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )
```

Not associated with either table

Summary

▪ Nested Queries

    ▪ IN, NOT IN, EXISTS, NOT EXISTS, op ANY and op ALL where op ϵ {<. <=, =, <>, >=, >}

    ▪ Re-writing INTERSECT using IN

    ▪ Re-writing EXCEPT using NOT IN

    ▪ Expressing the division operation using NOT EXISTS and EXCEPT (there are other ways to achieve that!)

▪ Other DML commands: INSERT (including bulk insertions), DELETE and UPDATE (for tables and views)

▪ Null values and inner vs. outer Joins

Indexing

Answering a range selection without making it slow like binary search.

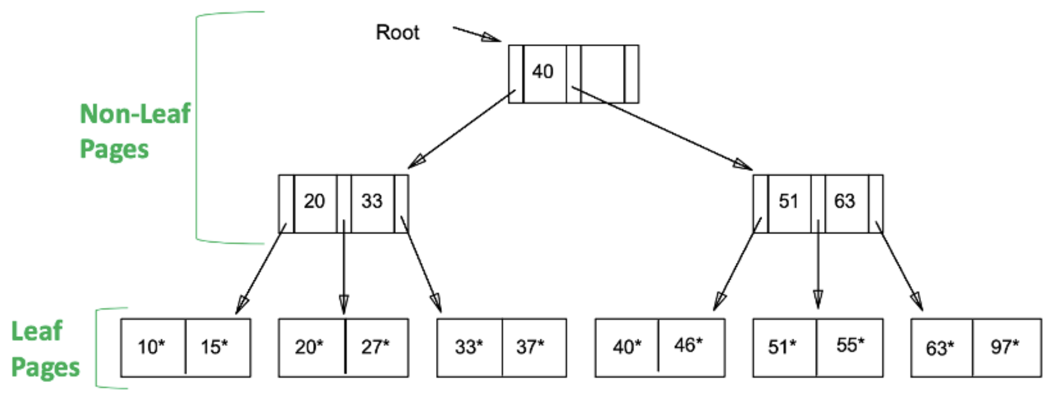
Data records Storage

1. K* is an actual data record with key *k*.
   - Leaf pages contain the actual data
2. K* is a <k, rid> pair, where rid is the record id of a data record with search key *k*.
   - Leaf pages contain the <k, rid> pairs and actual data records are stored in a separate file.
3. K* is a <k, rid-list> pair where rid-list is a list or rids of data records with search key *k*.
   - Leaf pages contain the <k, rid-list> pairs and actual data records are stored in a separate file.

Clustered Indexes: When the ordering of data records is the same as (or close to) the ordering of entries in some index. (Alternative 1)
Unclustered Indexes: When the ordering of data records differs from the ordering of entries in some index. (Alternative 2 and 3 unless data records are sorted on the search key field.

Clustered	***Nonclustered***
Only 1 allowed per table	Up to 249 allowed per table
Physically rearranges the data in the table to conform to the index constraints	Creates a separate list of key values with pointers to the location of the data in the data pages
For use on columns that are frequently searched for ranges of data	For use on columns that are searched for single values
For use on columns with low selectivity	For use on columns with high selectivity

Indexed Sequential Access Method (ISAM)



Insert/Delete

- The appropriate page is determined like for a search and then the entry is deleted/deleted.
- Overflow pages ONLY removed when becoming empty

Issues w/ ISAM

- Static structure , therefore once an ISAM file is created, insertions and deletions affect only the content of leaf pages
- No need to lock index-level pages during insertions/deletions (critical for concurrency)
- Long overflow chains can easily develop

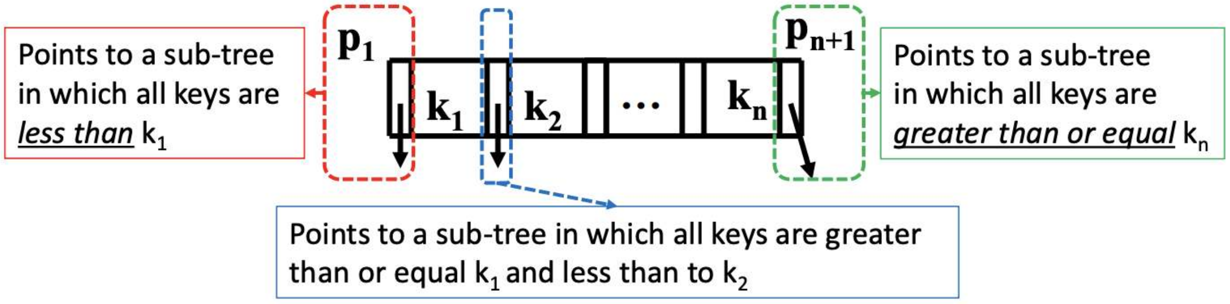
Choose ISAM if data distribution and size are relatively static

Dynamic Trees (B+ Trees)

Most successful dynamix index schemes: B+ Trees

B+ Trees Properties

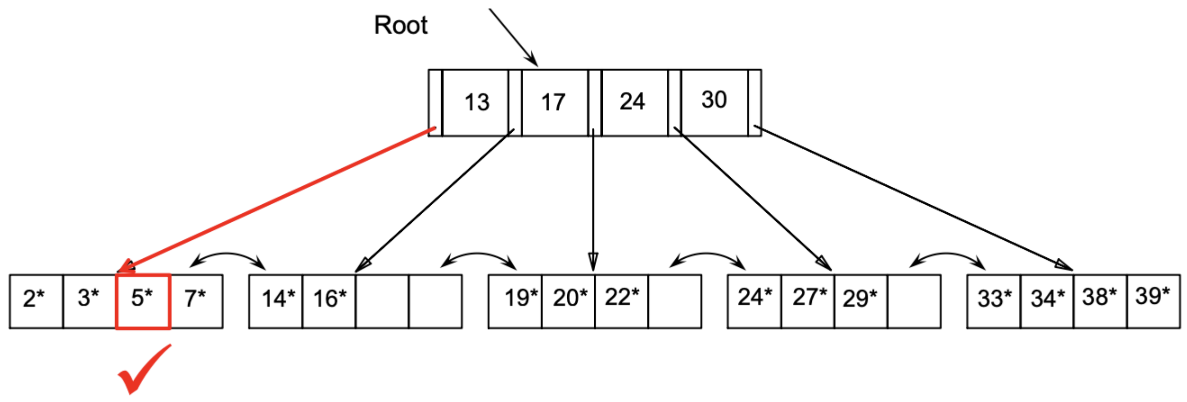
- Each node in a B+ tree of order ***d***.
    - Has at most **2d** keys
    - Has at least **d** keys (Except root)
    - All leaves are on the same level
    - Has exactly **n-1** keys if number of pointers is **n**


Points to a sub-tree in which all keys are *less than* k_1
p_1
k_1
k_2
…
k_n
p_{n+1}
Points to a sub-tree in which all keys are *greater than or equal* k_n
Points to a sub-tree in which all keys are greater than or equal k_1 and less than to k_2

Search

- Begins at root
- Key comparisons direct it to a leaf (as in ISAM)

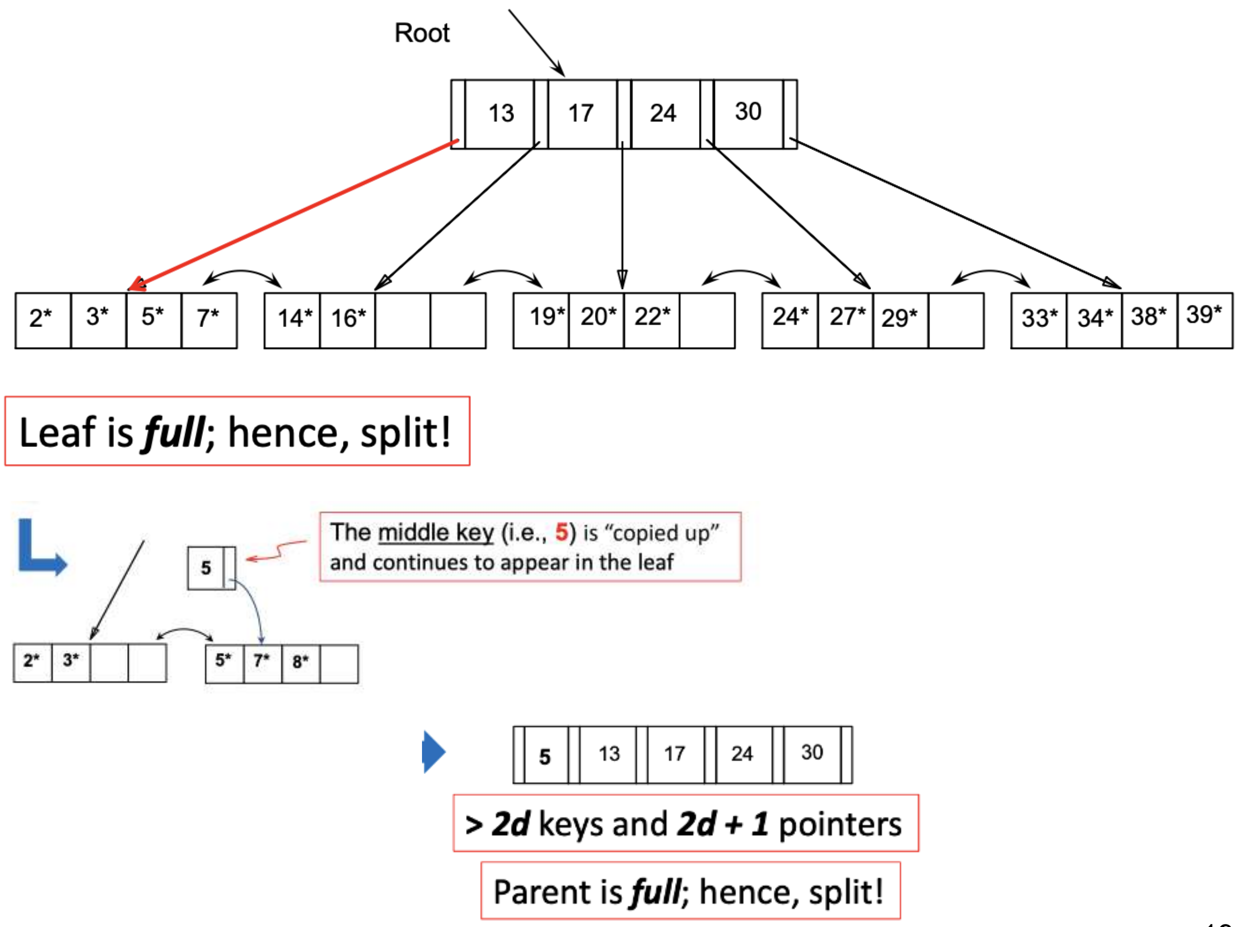
Eg. Search 5*



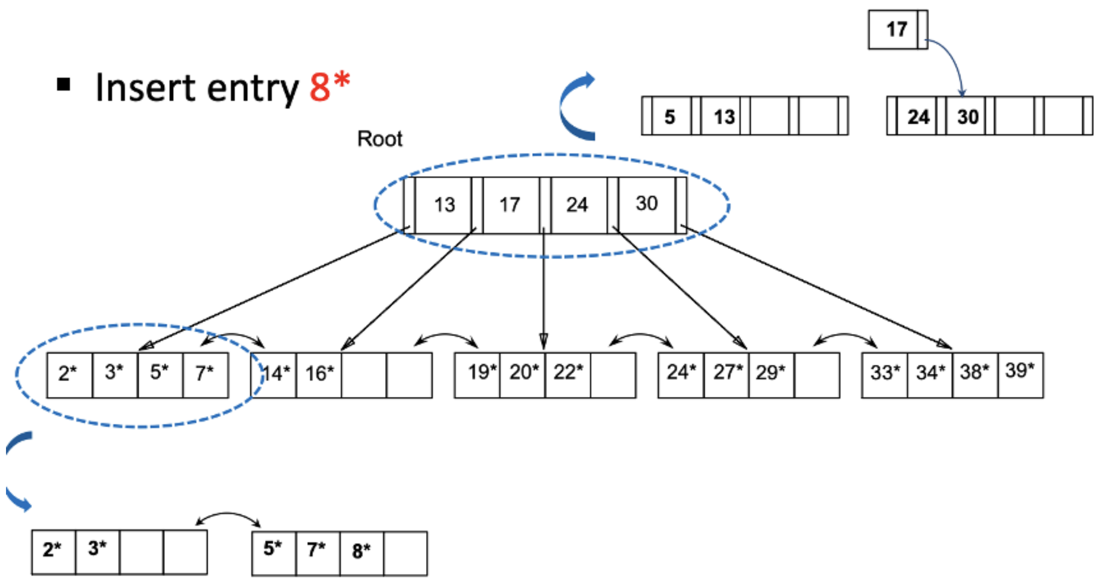
Inserting

- Find correct leaf L
- Put data entry onto L
    - If L has enough space => done
    - Else, split L into L and a new node L_2
        - Repartition entries evenly copying up the middle key
- If parent node overflow: Push up middle key (root split)
    - Increases the height of the tree
    -

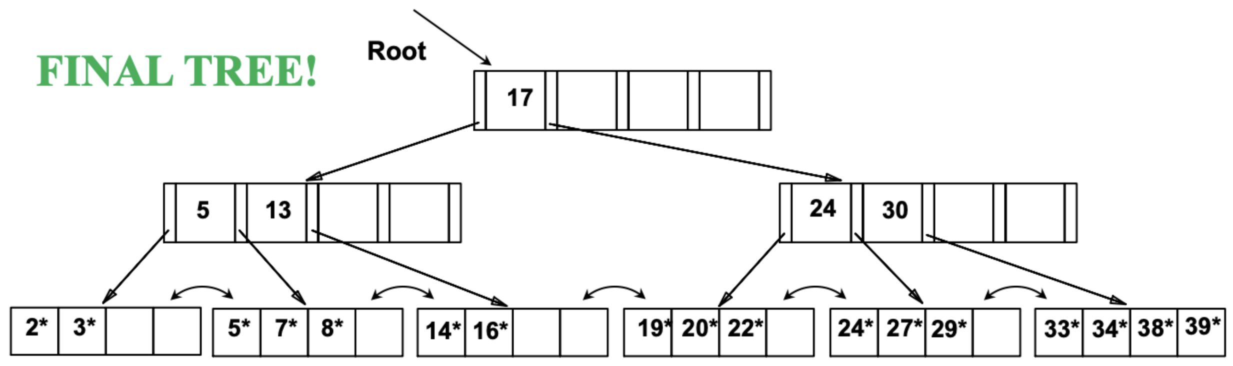
Eg. Insert 8*


Root
13
17
24
30
2*
3*
5*
7*
14*
16*
19*
20*
22*
24*
27*
29*
33*
34*
38*
39*
Leaf is full; hence, split!
The middle key (i.e., 5) is "copied up" and continues to appear in the leaf
5
2*
3*
5*
7*
8*
5
13
17
24
30
> 2d keys and 2d + 1 pointers
Parent is full; hence, split!

- Insert entry 8*



FINAL TREE!



Splitting vs. Redistributing

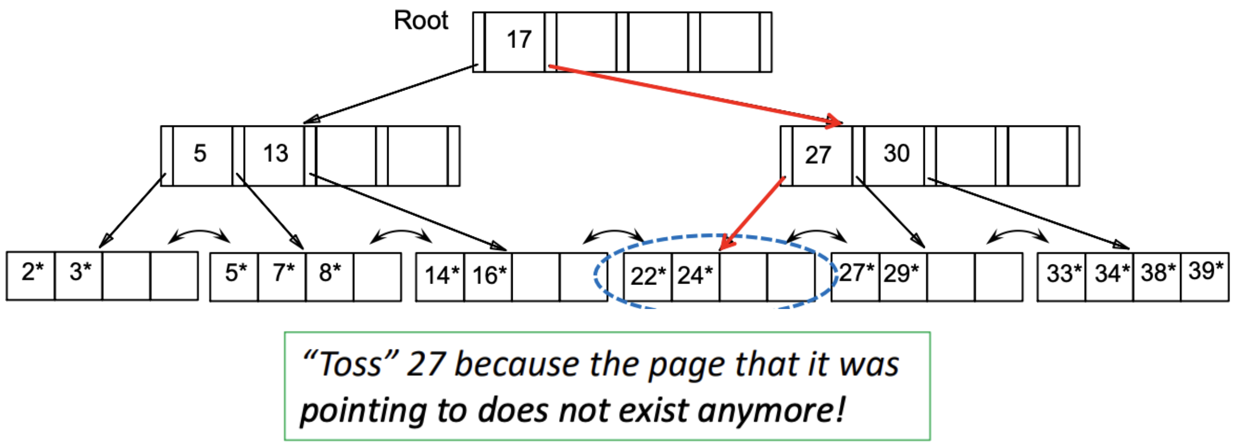
Leaf Node
- If a neighbor can spare an entry, re-distribute (does not increase I/O when checking)

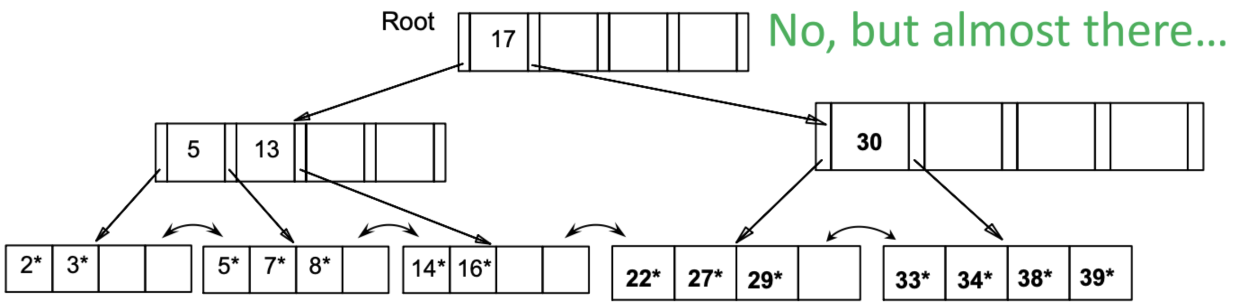
Non-Leaf Nodes
- Checking if redistribution is possible usually increases I/O
- Splitting non-leaf nodes typically pays off

Deleting Entries
- Start at root
- Find leaf L where entry belongs
- Remove the entry
    - If L at least half-full => done
    - If L underflows
        - Try to redistribute (Borrow from rich neighbor and copy up its lowest key)
        - Else, merge L and a poor sibling

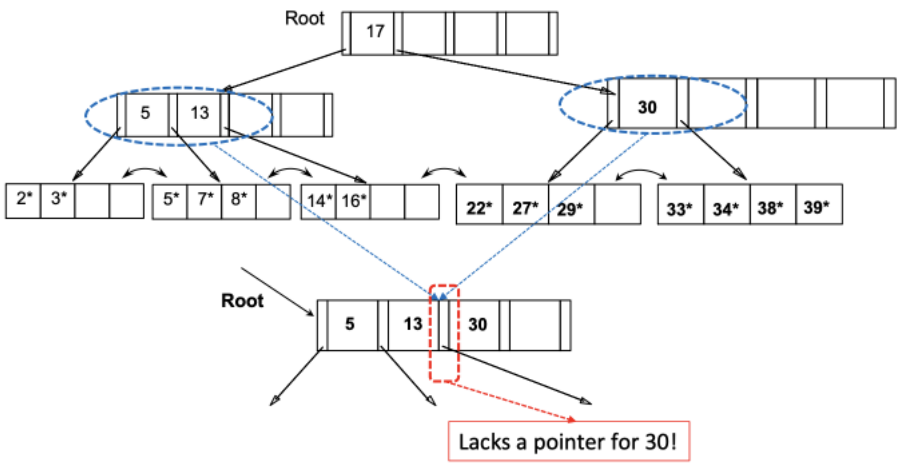
- Update parent
- Possibly merge recursively

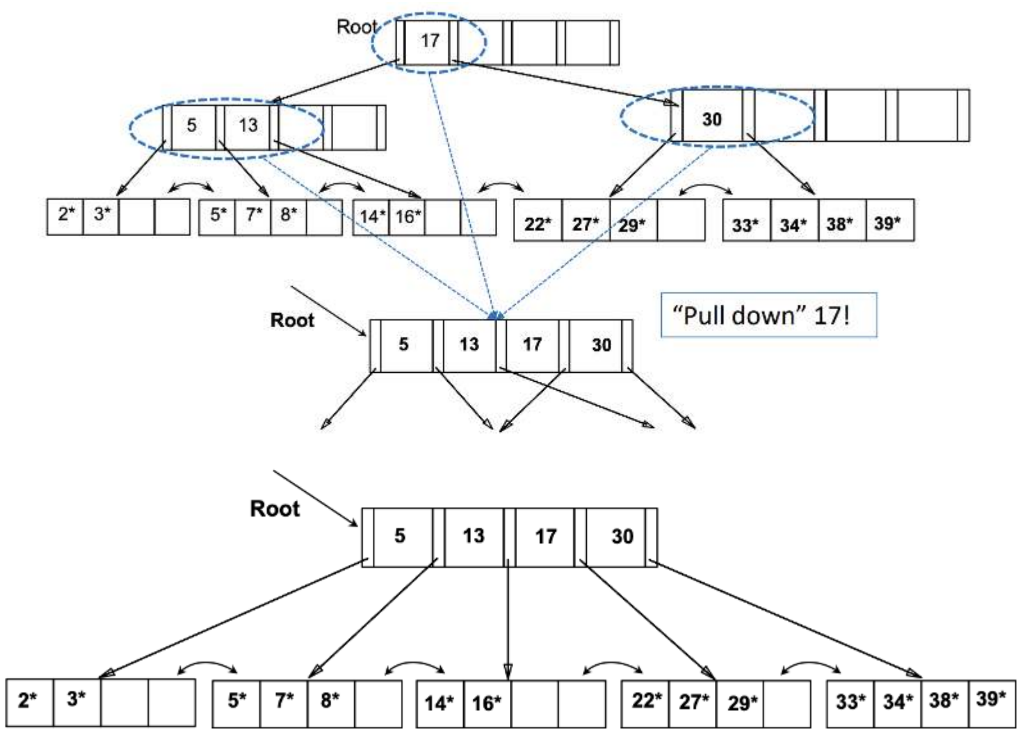
Eg. Delete 24*


Root
17
5
13
27
30
2*
3*
5*
7*
8*
14*
16*
22*
24*
27*
29*
33*
34*
38*
39*
"Toss" 27 because the page that it was pointing to does not exist anymore!



⇒ 30 alone entails an underflow, we must either redistribute or merge.
Its sibling is poor (5,13) so merge





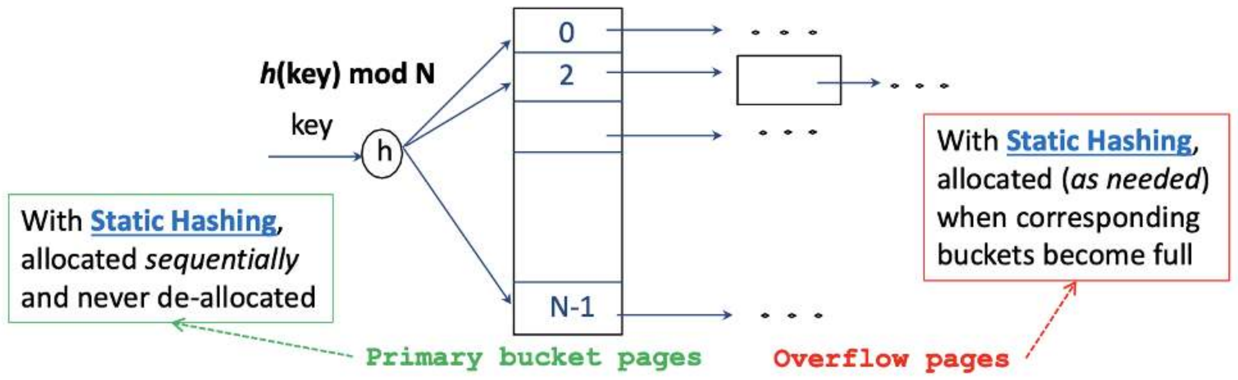
FINAL TREE!

Hash-Based Indexing

Doesn't support range searches
Very useful in implementing relational operators (eg joins)

Static Hashing
Hash structure is a generalization of the simpler notion of an ordinary array (arbitrary position examined in O(1))
Data entries can be any of the 3 alternatives and can be sorted in buckets to speed up searches.
Hash function : Used to map keys into a range of bucket number

- Used to identify the bucket to which a given key belongs and subsequently insert,delete or locate a data record.

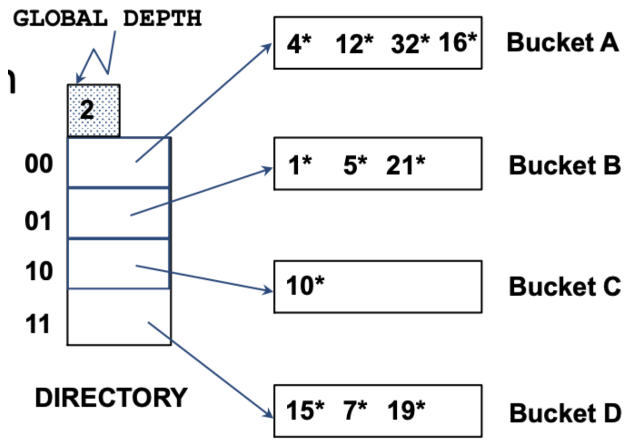

h(key) mod N
key
h
0
2
N-1
With Static Hashing, allocated sequentially and never de-allocated
With Static Hashing, allocated (as needed) when corresponding buckets become full
Primary bucket pages
Overflow pages

Search: 1 disk I/O
Insertion/Deletion : 2 disk I/Os

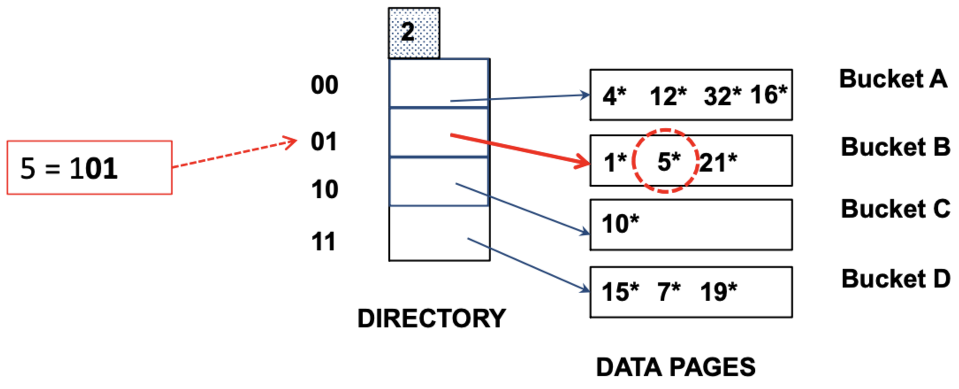
Issues:
- Fixed number of buckets: Like in ISAM
- Long overflow chain can develop easily and degrade performance

Extendible Hashing (Dynamic)
Uses a directory of pointes to buckets

The result of applying a hash function **h** is treated as a binary number and the last **d** bits are interpreted as an offset into the directory. **d** is the *global depth.* Kept as part of the header of the file

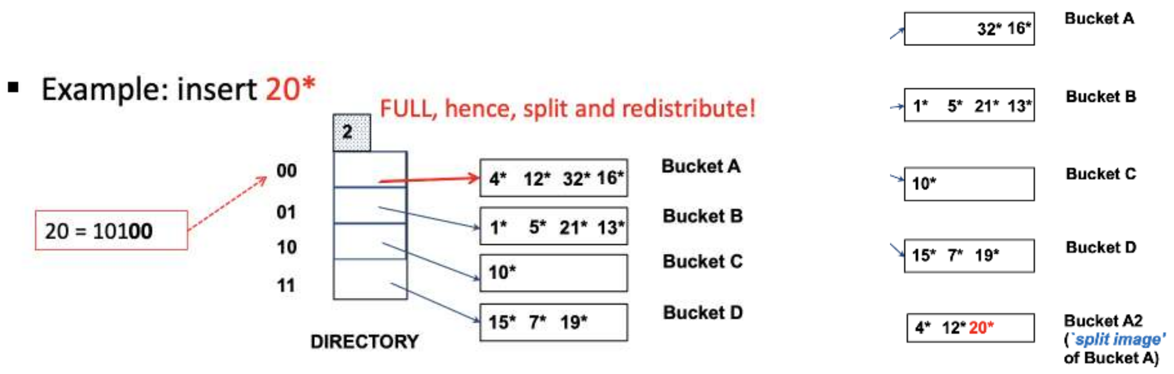


Eg. Search for 5*

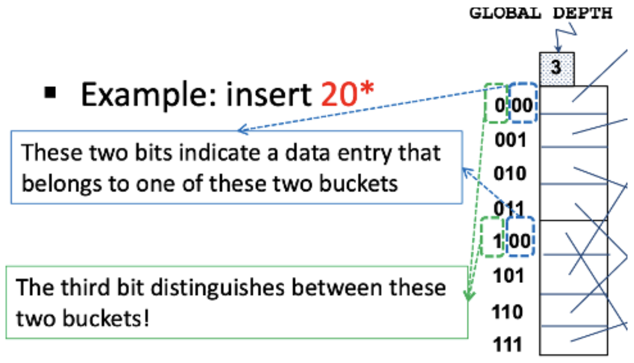


Insertion

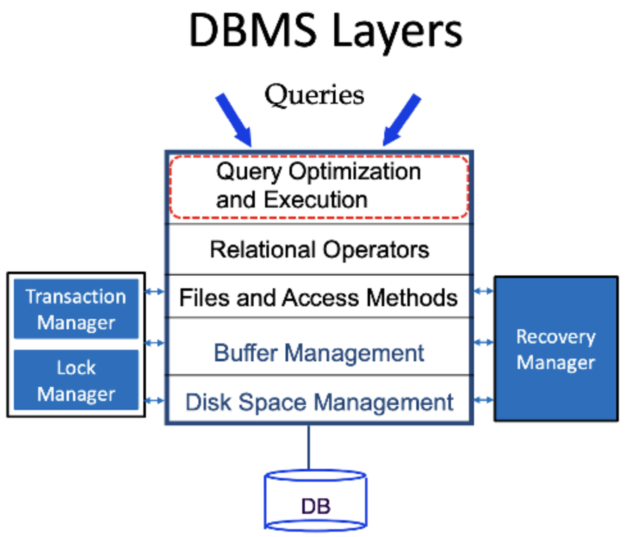
- Find appropriate bucket
- Split bucket if full and redistribute content (including new entry)
- Double the directory if necessary
- Insert the given entry



But now the local depth is the same as global, so we need to double directory and increase global depth

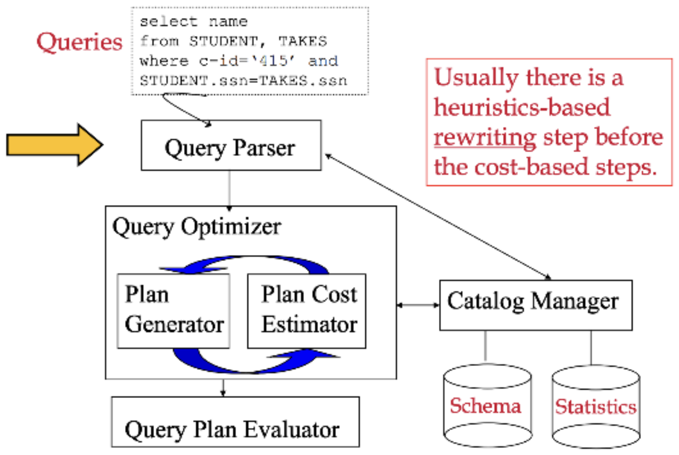

GLOBAL DEPTH
3
Example: insert 20*
These two bits indicate a data entry that belongs to one of these two buckets
The third bit distinguishes between these two buckets!
000
001
010
011
100
101
110
111

Query Optimization



Steps

1. Queries are parsed into internal forms (eg. parse trees)
2. Internal forms are transformed into 'canonical forms' (syntactical query optimization)
3. A <u>subset</u> of alternative plans are enumerated
4. Costs for alternative plans are estimated
5. The query evaluation plan with the *least estimated cost* is picked.



==> Cost Based Query Sub-Systems

To evaluate Queries:

- Need to estimate the costs of query plans, the query optimizer examines the system catalog and retrieves:
  - Information about the types and lengths of fields
  - Statistics about the referenced relations
  - Access paths (indexes) available for relations

- Schema and statistics components in catalog manager particularly inspected to find a good query evaluation plan

Catalog Manager

=> Schema

Information Stored in Schema:

- Information about tables and attributes
- Information about indices (eg. index structures)
- Information about users

Where:

- In tables (therefore it can be queried like any other tables)
- For example: Attribute_Cat (attr_name: string, rel_name: string; type: string; position: integer)

=> Statistics

Information stored in Statistics

- NTuples(R): # records for table R
- NPages(R): # pages for R
- NKeys(I): # distinct key values for index I
- INPages(I): # pages for index I
- IHeight(I): # levels for I
- ILow(I), IHigh(I): range of values for I
- ..

Stored to estimate plan costs and results sizes

SQL Blocks

Queries optimized by decomposing them into a collection of smaller units (blocks)

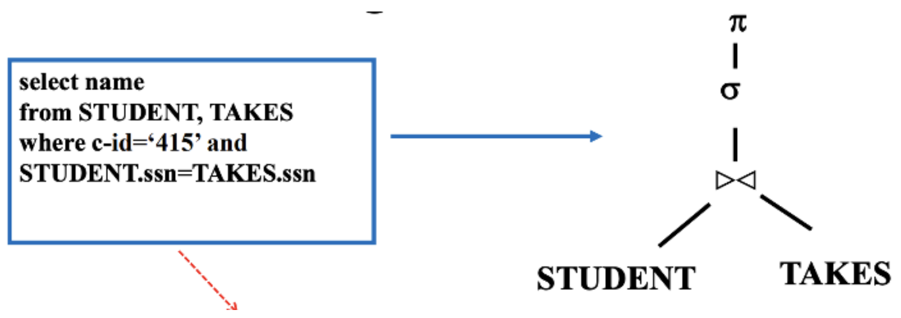
It is an SQL query with

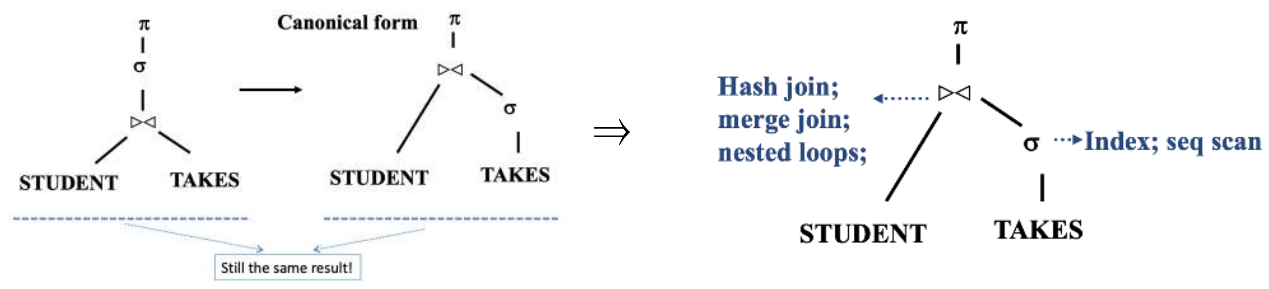
- NO nesting
- Exactly one SELECT and one FROM clause
- At most 1 WHERE, 1 GROUP BY and 1 HAVING clauses

Relation Algebra Trees

SQL block can be thought of an algebra expression with:

- A cross-product of all relations in the FROM clause
- Selections in the WHERE clause, used to select all columns of a specific tuple
- Projections in the SELECT clause, used to select specific columns


```
select name
from STUDENT, TAKES
where c-id='415' and
STUDENT.ssn=TAKES.ssn
```

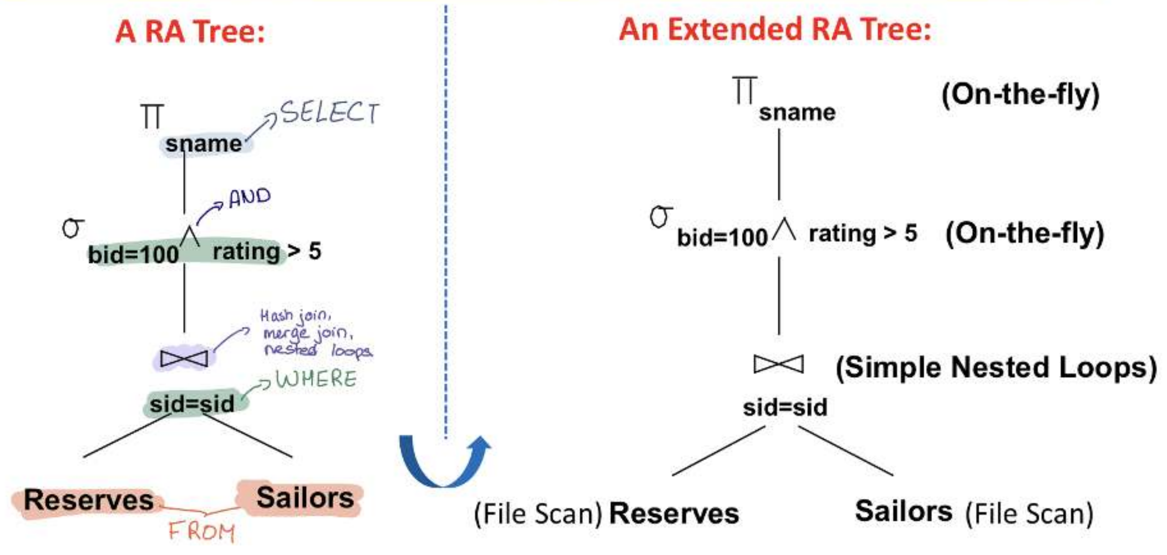


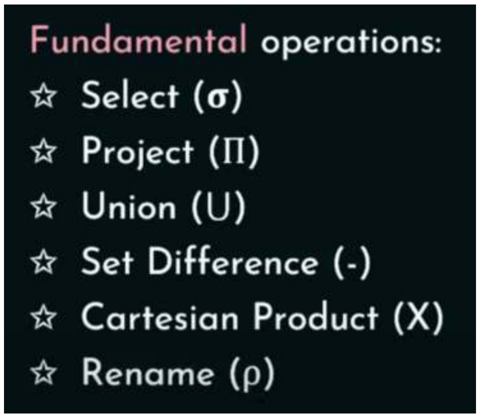
Query Evaluation Plan
Consists of an extended relational algebra tree.

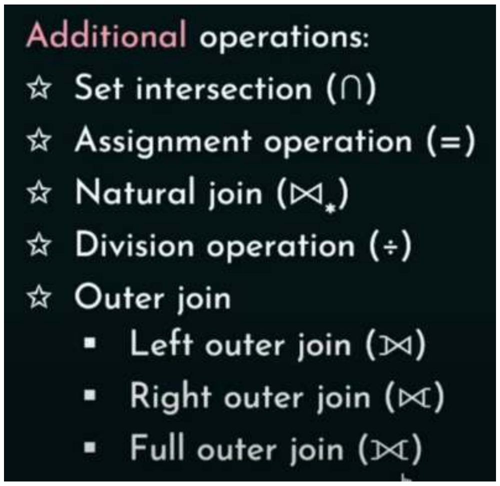
⇒A plan tree consists of annotations at each node indicating:
- Access methods to use for each relation
- Implementation method to use for each operator

Eg.
Query **Q**:
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
R.bid=100 AND S.rating>5

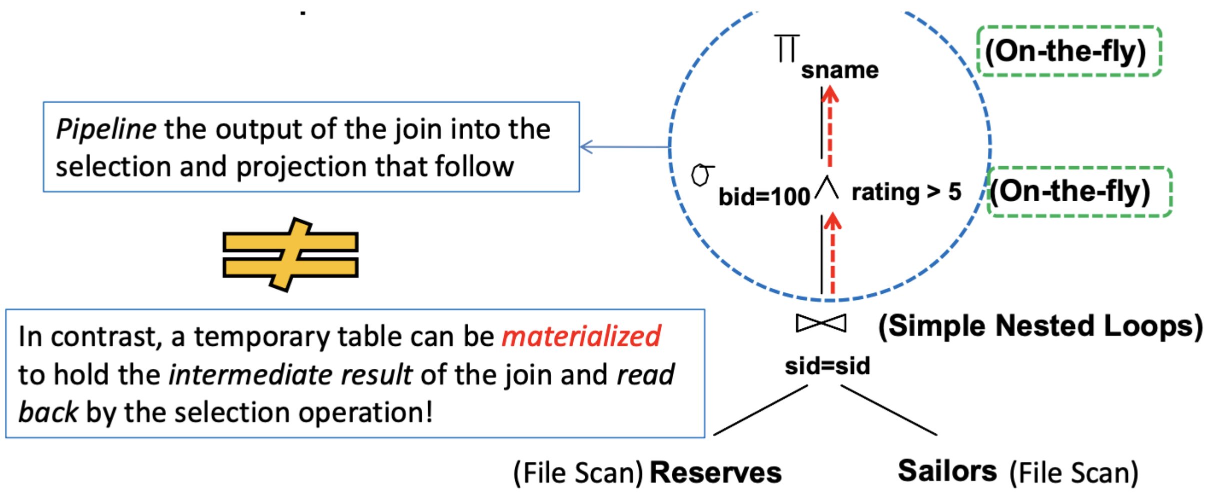
$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$$




Fundamental operations:
- ☆ Select (σ)
- ☆ Project (Π)
- ☆ Union (U)
- ☆ Set Difference (-)
- ☆ Cartesian Product (X)
- ☆ Rename (ρ)


Additional operations:
- ☆ Set intersection (∩)
- ☆ Assignment operation (=)
- ☆ Natural join ($\bowtie_*$)
- ☆ Division operation (÷)
- ☆ Outer join
  - Left outer join (⟕)
  - Right outer join (⟖)
  - Full outer join (⟗)

⇒ Pipelining vs. Materializing
When a query is composed of several operators, the result of one operator can sometimes be pipelined to another operator. Pipelining can save I/O cost

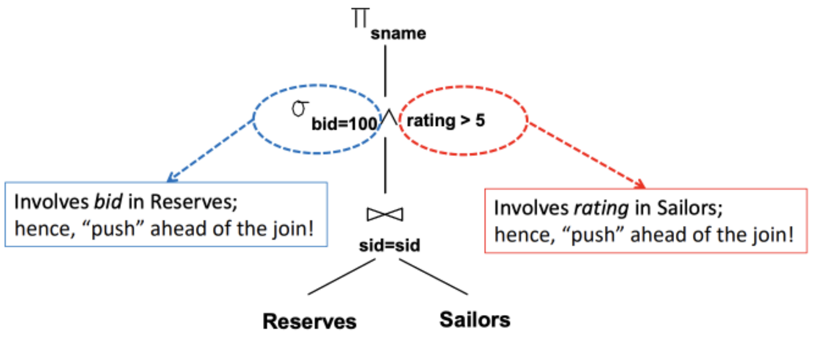

Pipeline the output of the join into the selection and projection that follow
≠
In contrast, a temporary table can be materialized to hold the intermediate result of the join and read back by the selection operation!
Π sname
(On-the-fly)
σ bid=100 ∧ rating > 5
(On-the-fly)
⋈ sid=sid
(Simple Nested Loops)
(File Scan) Reserves
Sailors (File Scan)

I/O cost of the **Q** plan

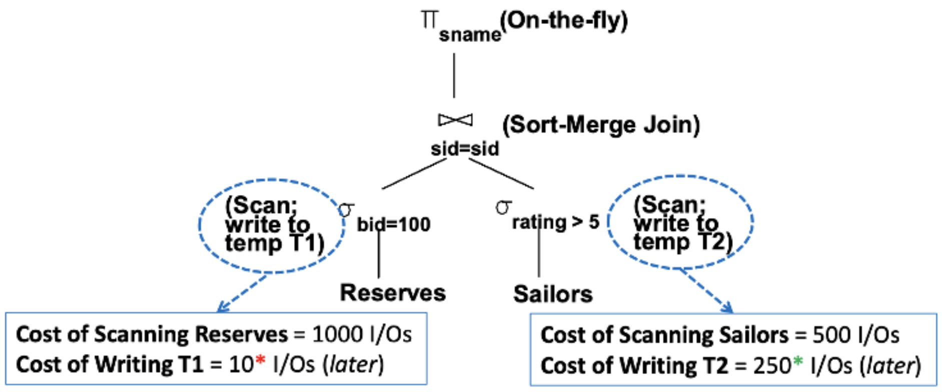
- ✓ The cost of the join is 1000 + 1000 * 500 = 501,000 I/Os (assuming page-oriented Simple NL join)
- ✓ The selection and projection are done on-the-fly; hence, do not incur additional I/Os

How can we Reduce the cost of a join?

⇒By reducing the sizes of input relations!


Π sname
σ bid=100 ∧ rating > 5
Involves bid in Reserves; hence, "push" ahead of the join!
Involves rating in Sailors; hence, "push" ahead of the join!
⋈ sid=sid
Reserves
Sailors

New Plan:


Π sname (On-the-fly)
⋈ sid=sid (Sort-Merge Join)
(Scan; write to temp T1)
σ bid=100
σ rating > 5
(Scan; write to temp T2)
Reserves
Sailors
Cost of Scanning Reserves = 1000 I/Os
Cost of Writing T1 = 10* I/Os (later)
Cost of Scanning Sailors = 500 I/Os
Cost of Writing T2 = 250* I/Os (later)

Relational Algebra Equivalences

A relational query optimizer uses relational algebra equivalences to identify many equivalent expressions for a given query.

Two relational algebra expressions over the same set of input relations are said to be equivalent if they produce the same results on all relations' instances

They allow us to:

- Push selections and projections ahead of joins
- Combine selections and cross-products into joins
- Choose different join orders

⇒ Selections

2 important equivalences involve selections

1. Cascading of selections

$$\sigma_{c1\wedge\ldots\wedge cn}(R) \equiv \sigma_{c1}(\ldots\ \sigma_{cn}(R))$$

Allows us to combine several selections into one selection

OR: Allows us to replace a selection with several smaller selections

2. Commutation of selections

$$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$$

Allows us to test selection conditions in either order

⇒ Projections

- Cascading of projections

$$\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{an}(R)))$$

This says that successively eliminating columns from a relation is equivalent to simply eliminating all but the columns retained by the final projection!

⇒ Cross-Products and Joins

1. Commutative Operations

$$(R \times S) \equiv (S \times R)$$
$$(R \bowtie S) \equiv (S \bowtie R)$$

This allows us to choose which relation to be the inner and which to be the outer!

2. Associative Operations

$$R \times (S \times T) \equiv (R \times S) \times T$$
$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$$

It follows: $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$

This says that regardless of the order in which the relations are considered, the final result is the same!

Note: We can commute a selection with a cross product or a join if the selection involves only attributes of one of the arguments to the cross-product or join

NoSQL Databases

NoSQL = Not Only SQL
Implies that more than one storage mechanism could be used based on the needs
Not singular, represents a class of products and a collection of concepts about data storage and manipulation.
NoSQL database systems: Represent a new generation of low-cost, high performance database software
Differences:

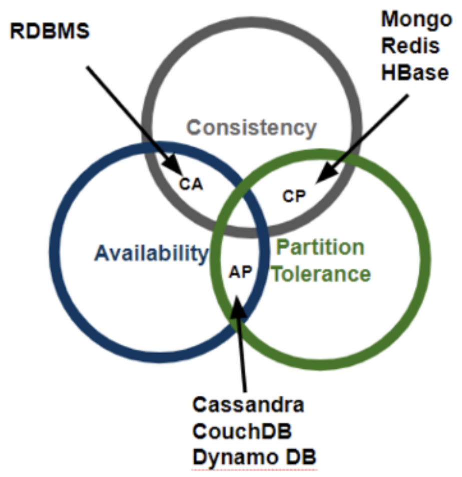
- Does Not use SQL as query language
- No fixed schema
- No join operations (requires fixed schemas and consistency)
- Allows distributed, Fault-tolerant architecture
- Allows linear scalability

Goals: Performance, reliability, consistency and enhancing search and read performance

Limitations:

- Consistency
    - Every node needs to always see the same data value at any given instance
- Availability
    - The system should continue to operate even if nodes in a cluster crash or some of the hardware/software parts are down at any time
- Partition Tolerance
    - The systems continue to operate in the presence of network partitions

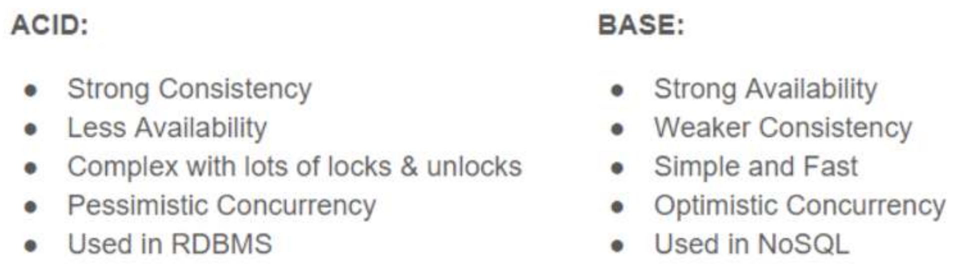
CAP Theorem: Any distributed database with shared data can have at most 2 of the 3 desirable properties: underlineConsistency, underlineAvailibility and underlinePartition Tolerance



Availability + Partition Tolerance = Not Consistent
Consistency + Partition Tolerance = Not available
Availability + Consistency = Not Partition

BASE: (Basically Available, Soft State and Eventually Consistent)
Transaction control models, like Acid used in RDBMS
Difference is the amount of effort required by application developers and the location of the transactional controls.
Main focus is **availability** (while it is integrity and consistency for ACID).
Optimistic, simpler and faster than ACID. (No locking/unlocking)

Basic Availability: System can be temporarily inconsistent so that transaction are manageable and all information and services are “Basically Available”
Soft State: Some inaccuracies are allowed and data may change during usage to reduce the amount of used resources for the sake of availability.
Eventual Consistency: At a certain point when all service logic is executed, the system turns into a consistent state.



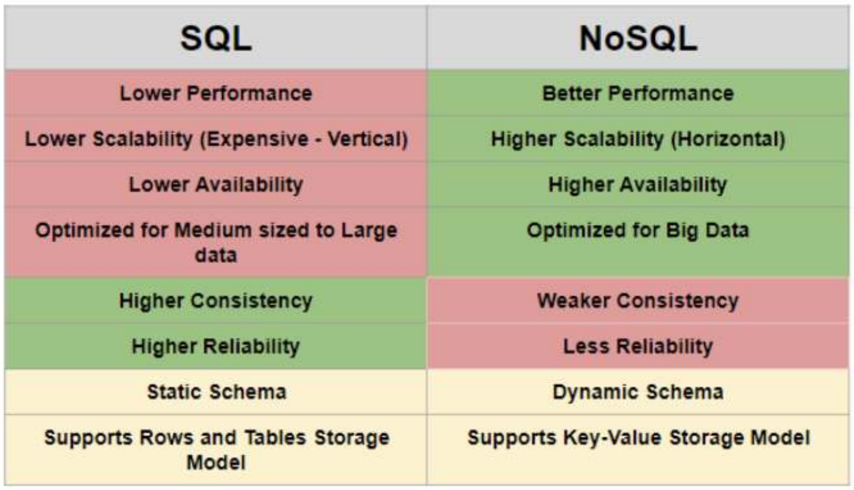
ACID:	BASE:
• Strong Consistency	• Strong Availability
• Less Availability	• Weaker Consistency
• Complex with lots of locks & unlocks	• Simple and Fast
• Pessimistic Concurrency	• Optimistic Concurrency
• Used in RDBMS	• Used in NoSQL

Advantages of NoSQL:

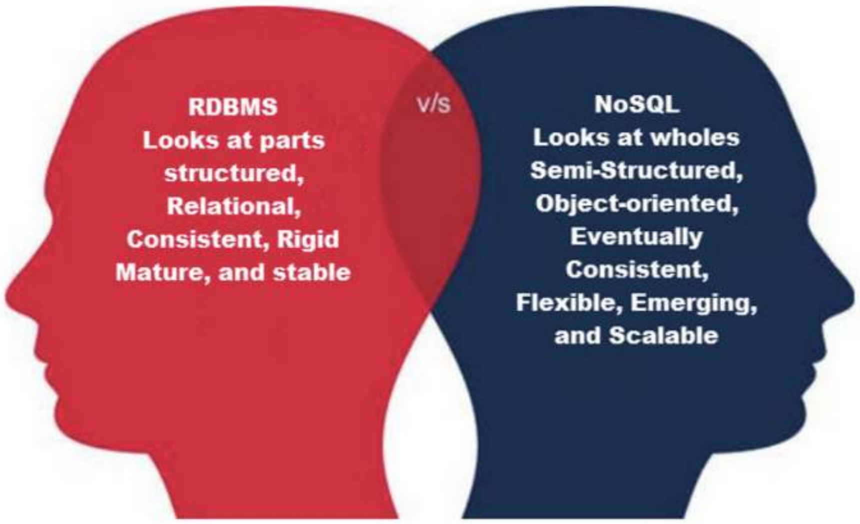
- Simple and Flexible Structures (schema free)
- Based on Key-Value Pairs
- Allow storage of serialized objects in the database
- Open source databases don't need expensive licensing fees.
- Can run on inexpensive software.
- Expansion is always cheaper and easier.
- Depends on horizontal scaling by distributing the load over more nodes

Disadvantages:

- Don't support the reliability features of relational databases and ACID transactions control systems.
- They trade consistency for the sake of availability, performance and scalability
- Incompatible with SQL queries (more complexity by manual querying languages)
- Don't support Joins/Group By/Order by/ Acid transactions.

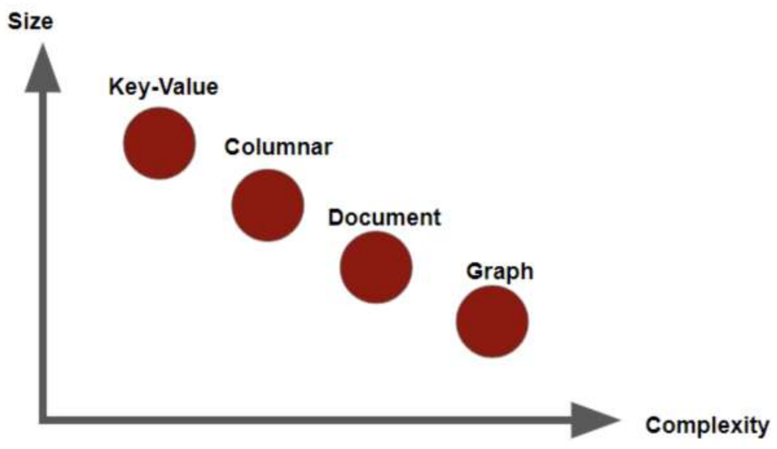


SQL	NoSQL
Lower Performance	Better Performance
Lower Scalability (Expensive - Vertical)	Higher Scalability (Horizontal)
Lower Availability	Higher Availability
Optimized for Medium sized to Large data	Optimized for Big Data
Higher Consistency	Weaker Consistency
Higher Reliability	Less Reliability
Static Schema	Dynamic Schema
Supports Rows and Tables Storage Model	Supports Key-Value Storage Model



Classification of NoSQL databases

- Column
  - Hybrid of Key-valued and RDBMS
  - Data stored in columns instead of rows
  - Consist of one or more column families that group certain columns in the database
  - Each key is used to identify and point to one family
  - Data separated by commas
  - Fast data aggregation
- Key-Value
- Graph
  - Graph includes set of nodes, each representing an object
  - Some pairs of objects are connected by edges representing relations between objects
  - Typical For social networking applications
  - Focus on relations between object
  - Powerful for graph queries
  - Eg neo4j
- Document
  - Similar to Key-value (same advantages/disadvantages and schema-less)
  - Value of each key stored in the database is a document
  - Document themselves can contain many different key-value pairs
  - Documents can be indexed (which leads to outperformance of traditional file systems)
  - Eg. mongodb



Cypher

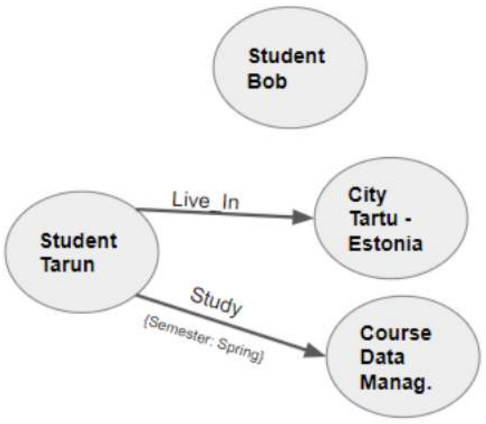
- Declarative graph query language
- Allows expressive and efficient querying and updating of a property graph
- Relatively simple, but powerful language

Eg.

CREATE (a:Studentname:’Tarun’) -> (NODE)
CREATE (a:Studentname:’Bob’)-> (NODE)
CREATE (b:Cityname:’Tartu’, located In:’Estonia’)-> (NODE)
CREATE (c:Coursename:’Data Management’)-> (NODE)

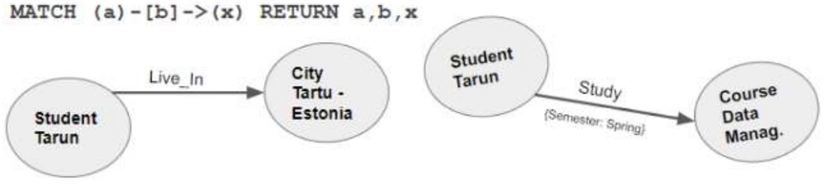
CREATE (b)<-[:Live In]-(a)-[:Studysemester:’spring’]->(c) -> (EDGE)

```
MATCH (m:Movie {title:"A Few Good Men"})  WITH m
MATCH (m) <- [:ACTED□IN] - (p:Person)
return m,p
```



Match will retrieve all nodes with specific relation
Eg.

```
MATCH (a)-[b]->(x) RETURN a,b,x
```



Merge will merge some nodes together after matching
Set command will update/insert some data to a node/edge after matching
Delete command will delete an edge/node after matching

Apache

For column databases.
Scan will retrieve all data from a given table
Get will retrieve data from a given table based on some parameters.
Put command will insert/overwrite/update entries in a tabler

Eg.
create ’student’, ’personal data’, ’courses’
Put ’student’,’87787’, ’personal data:name’,’Tarun’
delete ’students’, ’87787’, ’courses:spring’

Indexing Structures For NoSQL databases
Indexing ⇒ The process of associating a key with the location of a corresponding data record in a Database management system.

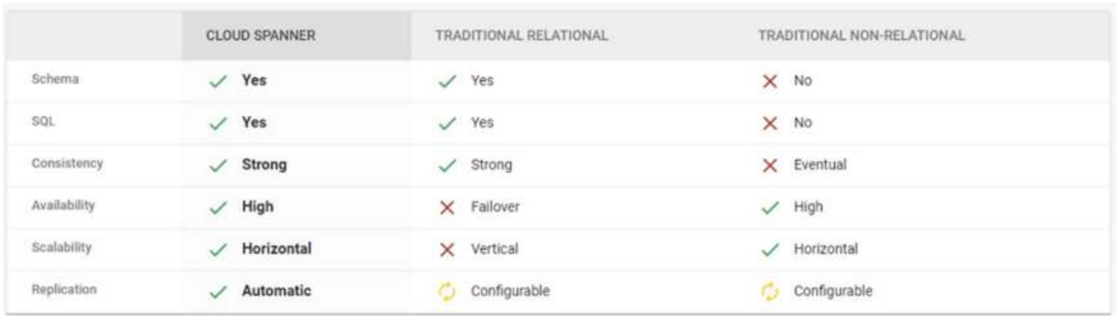
3 most common processes:

- B-Tree indexing
- T-Tree indexing
- O2-Tree Indexing

Cloud Spanner

Google's globally distributed NewSQL database

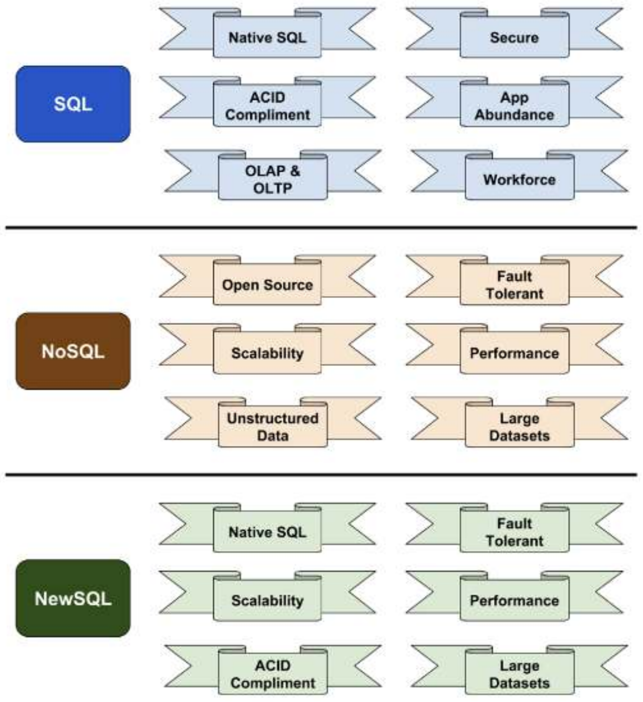
- Only entreprise-grade, globally distributed and strongly consistent database service built for the cloud specifically to combine the benefits of relational databases with non-relational horizontal scaling.
- Delivers high-performance transactions and strong consistency.



	CLOUD SPANNER	TRADITIONAL RELATIONAL	TRADITIONAL NON-RELATIONAL
Schema	✓ Yes	✓ Yes	✗ No
SQL	✓ Yes	✓ Yes	✗ No
Consistency	✓ Strong	✓ Strong	✗ Eventual
Availability	✓ High	✗ Failover	✓ High
Scalability	✓ Horizontal	✗ Vertical	✓ Horizontal
Replication	✓ Automatic	Configurable	Configurable

NewSQL:

- Seeks to provide the same scalability performance of noSQL and maintain ACID properties of traditional db systems.
- They are relational databases


SQL
Native SQL
Secure
ACID Compliment
App Abundance
OLAP & OLTP
Workforce
NoSQL
Open Source
Fault Tolerant
Scalability
Performance
Unstructured Data
Large Datasets
NewSQL
Native SQL
Fault Tolerant
Scalability
Performance
ACID Compliment
Large Datasets