



*W4111 – Introduction to Databases
Section 002, Fall 2023
Lecture 8:
Finishing ER, Relational, SQL
Starting Module II and NoSQL*

Contents

Codd's Rules Discussion

Codd's 12 Rules

Rule 1: Information Rule

The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.

Rule 2: Guaranteed Access Rule

URL Like

Every single data element (value) is guaranteed to be accessible logically with a combination of table-name, primary-key (row value), and attribute-name (column value).

No other means, such as pointers, can be used to access data.

Rule 3: Systematic Treatment of NULL Values

The NULL values in a database must be given a systematic and uniform treatment. This is a very important rule because a NULL can be interpreted as one the following – data is missing, data is not known, or data is not applicable.

Rule 4: Active Online Catalog

The structure description of the entire database must be stored in an online catalog, known as data dictionary, which can be accessed by authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

Rule 5: Comprehensive Data Sub-Language Rule

A database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation.

Rule 6: View Updating Rule

All the views of a database, which can theoretically be updated, must also be updatable by the system.

Codd's 12 Rules

Rule 7: High-Level Insert, Update, and Delete Rule

A database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to yield sets of data records.

Rule 8: Physical Data Independence

The data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications.

Rule 9: Logical Data Independence

The logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. **For example, if two tables are merged or one is split into two different tables, there should be no impact or change on the user application. This is one of the most difficult rule to apply.**

Rule 10: Integrity Independence

A database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface.

Rule 11: Distribution Independence

The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems.

Rule 12: Non-Subversion Rule

If a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.

ER Modeling Continued



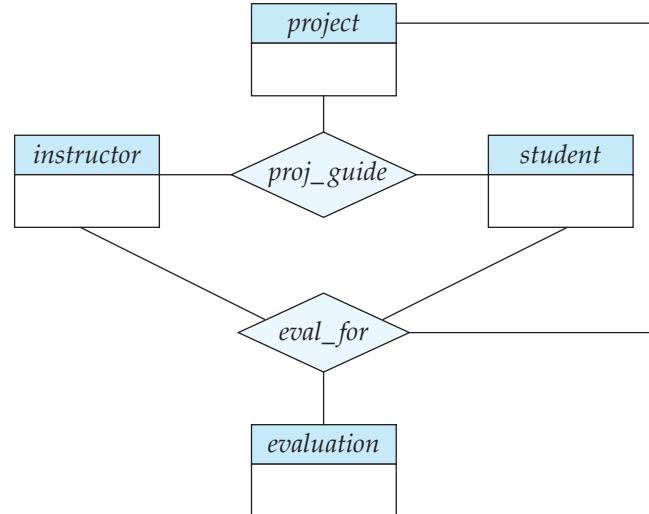
Mapping Cardinality Constraints

- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
 - One to one
 - One to many
 - Many to one
 - Many to many



Aggregation

- Consider the ternary relationship *proj_guide*, which we saw earlier
- Suppose we want to record evaluations of a student by a guide on a project





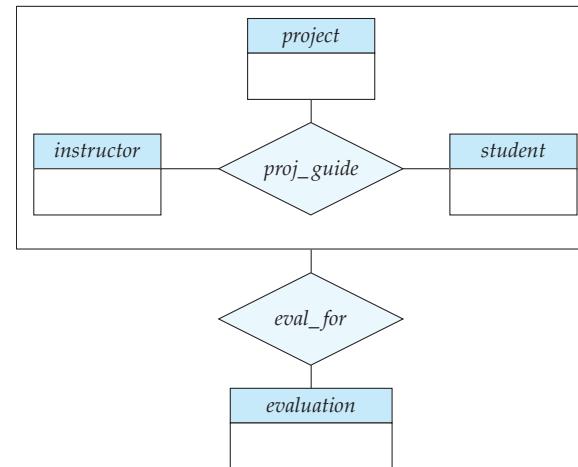
Aggregation (Cont.)

- Relationship sets *eval_for* and *proj_guide* represent overlapping information
 - Every *eval_for* relationship corresponds to a *proj_guide* relationship
 - However, some *proj_guide* relationships may not correspond to any *eval_for* relationships
 - So we can't discard the *proj_guide* relationship
- Eliminate this redundancy via *aggregation*
 - Treat relationship as an abstract entity
 - Allows relationships between relationships
 - Abstraction of relationship into new entity



Aggregation (Cont.)

- Eliminate this redundancy via *aggregation* without introducing redundancy, the following diagram represents:
 - A student is guided by a particular instructor on a particular project
 - A student, instructor, project combination may have an associated evaluation





Reduction to Relational Schemas

- To represent aggregation, create a schema containing
 - Primary key of the aggregated relationship,
 - The primary key of the associated entity set
 - Any descriptive attributes
- In our example:
 - The schema *eval_for* is:
$$\text{eval_for} (s_ID, project_id, i_ID, evaluation_id)$$
 - The schema *proj_guide* is redundant.



Specialization

- Top-down design process; we designate sub-groupings within an entity set that are distinctive from other entities in the set.
- These sub-groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
- Depicted by a *triangle* component labeled ISA (e.g., *instructor* “is a” *person*).
- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.

Functions, Procedures, Triggers

Summary

Comparison

comparing triggers, functions, and procedures

	triggers	functions	stored procedures
change data	yes	no	yes
return value	never	always	sometimes
how they are called	reaction	in a statement	exec

lynda.com

Comparison – Some Details

A *trigger* has capabilities like a procedure, except ...

- You do not call it. The DB engine calls it before or after an INSERT, UPDATE, DELETE.
- The inputs are the list of incoming new, modified rows.
- The outputs are the modified versions of the new or modified rows.

Sr.No.	User Defined Function	Stored Procedure
1	Function must return a value.	Stored Procedure may or not return values.
2	Will allow only Select statements, it will not allow us to use DML statements.	Can have select statements as well as DML statements such as insert, update, delete and so on
3	It will allow only input parameters, doesn't support output parameters.	It can have both input and output parameters.
4	It will not allow us to use try-catch blocks.	For exception handling we can use try catch blocks.
5	Transactions are not allowed within functions.	Can use transactions within Stored Procedures.
6	We can use only table variables, it will not allow using temporary tables.	Can use both table variables as well as temporary table in it.
7	Stored Procedures can't be called from a function.	Stored Procedures can call functions.
8	Functions can be called from a select statement.	Procedures can't be called from Select/Where/Having and so on statements. Execute/Exec statement can be used to call/execute Stored Procedure.
9	A UDF can be used in join clause as a result set.	Procedures can't be used in Join clause

Triggers



Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
 - Specify the conditions under which the trigger is to be executed.
 - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
 - Syntax illustrated here may not work exactly on your database system; check the system manuals



Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
 - For example, **after update of takes on grade**
- Values of attributes before and after an update can be referenced
 - **referencing old row as** : for deletes and updates
 - **referencing new row as** : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
    when (nrow.grade = ' ')
begin atomic
    set nrow.grade = null;
end;
```



Trigger to Maintain credits_earned value

- **create trigger** *credits_earned* **after update of** *takes* **on** (*grade*)
referencing new row as *nrow*
referencing old row as *orow*
for each row
when *nrow.grade* \neq 'F' **and** *nrow.grade* **is not null**
and (*orow.grade* = 'F' **or** *orow.grade* **is null**)
begin atomic
 update *student*
 set *tot_cred*= *tot_cred* +
 (**select** *credits*
 from *course*
 where *course.course_id*= *nrow.course_id*)
 where *student.id* = *nrow.id*;
end;



Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
 - Use **for each statement** instead of **for each row**
 - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
 - Can be more efficient when dealing with SQL statements that update a large number of rows



When Not To Use Triggers

- Triggers were used earlier for tasks such as
 - Maintaining summary data (e.g., total salary of each department)
 - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
 - Databases today provide built in materialized view facilities to maintain summary data
 - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
 - Define methods to update fields
 - Carry out actions as part of the update methods instead of through a trigger



When Not To Use Triggers (Cont.)

- Risk of unintended execution of triggers, for example, when
 - Loading data from a backup copy
 - Replicating updates at a remote site
 - Trigger execution can be disabled before such actions.
- Other risks with triggers:
 - Error leading to failure of critical transactions that set off the trigger
 - Cascading execution

Security

Security Concepts (Terms from Wikipedia)

- Definitions:
 - “A (digital) identity is information on an entity used by computer systems to represent an external agent. That agent may be a person, organization, application, or device.”
 - “Authentication is the act of proving an assertion, such as the identity of a computer system user. In contrast with identification, the act of indicating a person or thing's identity, authentication is the process of verifying that identity.”
 - “Authorization is the function of specifying access rights/privileges to resources, ... More formally, "to authorize" is to define an access policy. ... During operation, the system uses the access control rules to decide whether access requests from (authenticated) consumers shall be approved (granted) or disapproved.
 - “Within an organization, roles are created for various job functions. The permissions to perform certain operations are assigned to specific roles. Members or staff (or other system users) are assigned particular roles, and through those role assignments acquire the permissions needed to perform particular system functions.”
 - “In computing, privilege is defined as the delegation of authority to perform security-relevant functions on a computer system. A privilege allows a user to perform an action with security consequences. Examples of various privileges include the ability to create a new user, install software, or change kernel functions.”
- SQL and relational database management systems implementing security by:
 - Creating identities and authentication policies.
 - Creating roles and assigning identities to roles.
 - Granting and revoking privileges to/from roles and identities.



Authorization

- We may assign a user several forms of authorizations on parts of the database.
 - **Read** - allows reading, but not modification of data.
 - **Insert** - allows insertion of new data, but not modification of existing data.
 - **Update** - allows modification, but not deletion of data.
 - **Delete** - allows deletion of data.
- Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.



Authorization (Cont.)

- Forms of authorization to modify the database schema
 - **Index** - allows creation and deletion of indices.
 - **Resources** - allows creation of new relations.
 - **Alteration** - allows addition or deletion of attributes in a relation.
 - **Drop** - allows deletion of relations.



Authorization Specification in SQL

- The **grant** statement is used to confer authorization
grant <privilege list> on <relation or view > to <user list>
- <user list> is:
 - a user-id
 - **public**, which allows all valid users the privilege granted
 - A role (more on this later)
- Example:
 - **grant select on department to Amit, Satoshi**
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).



Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
 - Example: grant users U_1 , U_2 , and U_3 **select** authorization on the *instructor* relation:

```
grant select on instructor to U1, U2, U3
```
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges



Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.
revoke <privilege list> on <relation or view> from <user list>
- Example:
revoke select on student from U₁, U₂, U₃
- <privilege-list> may be **all** to revoke all privileges the revoker may hold.
- If <revoker-list> includes **public**, all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantors, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.



Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
- To create a role we use:
 - **create a role <name>**
- Example:
 - **create role instructor**
- Once a role is created we can assign “users” to the role using:
 - **grant <role> to <users>**



Roles Example

- **create role** instructor;
- **grant** *instructor* **to** Amit;
- Privileges can be granted to roles:
 - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
 - **create role** teaching_assistant
 - **grant** *teaching_assistant* **to** *instructor*;
 - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
 - **create role** dean;
 - **grant** *instructor* **to** *dean*;
 - **grant** *dean* **to** Satoshi;



Authorization on Views

- `create view geo_instructor as
(select *
from instructor
where dept_name = 'Geology');`
- `grant select on geo_instructor to geo_staff`
- Suppose that a `geo_staff` member issues
 - `select *
from geo_instructor;`
- What if
 - `geo_staff` does not have permissions on `instructor`?
 - Creator of view did not have some permissions on `instructor`?



Other Authorization Features

- **references** privilege to create foreign key
 - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
 - Why is this required?
- transfer of privileges
 - **grant select on** *department* **to** Amit **with grant option**;
 - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
 - **revoke select on** *department* **from** Amit, Satoshi **restrict**;
 - And more!

Note:

- Like in many other cases, SQL DBMS have product specific variations.

Switch to notebook.

Some Advanced SQL

Recursive Queries

I do not recommend this.

Module II Kickoff

Course Modules – Reminder

Course Overview

Each section of W4111 is slightly different based on student interest and professor's focus. There is a common, core syllabus. Professors cover topics in different orders and grouping based on teaching style.

This section of W4111 has four modules:

- **Foundational concepts (50% of semester):** This module covers concepts like data models, relational model, relational databases and applications, schema, normalization, ... The module focuses on the relational model and relational databases. The concepts are critical and foundational for all types of databases and data centric applications.
- **Database management system architecture and implementation (10%):** This module covers the software architecture, algorithms and implementation techniques that allow [databases management systems](#) to deliver functions. Topics include memory hierarchy, storage systems, caching/buffer pools, indexes, query processing, query optimization, transaction processing, isolation and concurrency control.
- **NoSQL – “Not Only SQL” databases (20%):** This module provides motivation for [“NoSQL”](#) data models and databases, and covers examples and use cases. The module also includes cloud databases and databases-as-a-service.
- **Data Enabled Decision Support (20%):** This module covers data warehouses, data import and cleanse, OLAP, Pivot Tables, Star Schema, reporting and visualization, and provides an overview of analysis techniques, e.g. clustering, classification, analysis, mining.

Module II – DBMS Architecture and Implementation Overview and Reminder

Module II – DBMS Architecture and

What is a database? In essence a database is nothing more than a collection of information that exists over a long period of time, often many years. In common parlance, the term *database* refers to a collection of data that is managed by a DBMS. The DBMS is expected to:

- 
1. Allow users to create new databases and specify their *schemas* (logical structure of the data), using a specialized *data-definition language*.

Covered for the relational model.

Database Systems: The Complete Book (2nd Edition)

by [Hector Garcia-Molina](#) (Author), [Jeffrey D. Ullman](#) (Author), [Jennifer Widom](#) (Author)

Module II – DBMS Architecture and

- 
- 
2. Give users the ability to *query* the data (a “query” is database lingo for a question about the data) and modify the data, using an appropriate language, often called a *query language* or *data-manipulation language*.
 3. Support the storage of very large amounts of data — many terabytes or more — over a long period of time, allowing efficient access to the data for queries and database modifications.
 4. Enable *durability*, the recovery of the database in the face of failures, errors of many kinds, or intentional misuse.
 5. Control access to data from many users at once, without allowing unexpected interactions among users (called *isolation*) and without actions on the data to be performed partially but not completely (called *atomicity*).

Database Systems: The Complete Book (2nd Edition)

by [Hector Garcia-Molina](#) (Author), [Jeffrey D. Ullman](#) (Author), [Jennifer Widom](#) (Author)

Purpose of Database Systems

In the early days, database applications were built directly on top of file systems, which leads to:

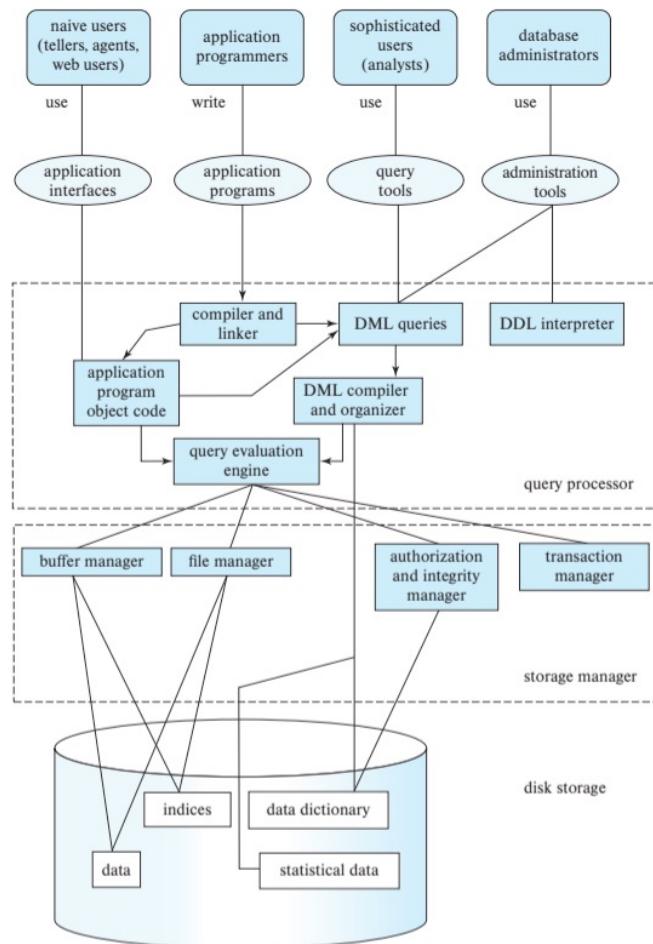
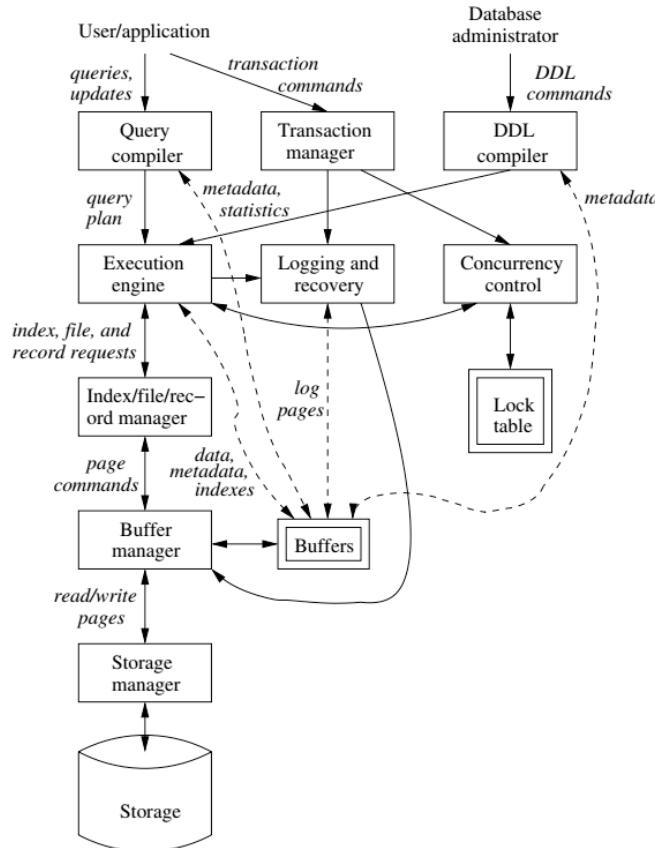
- Data redundancy and inconsistency: data is stored in multiple file formats resulting in duplication of information in different files
- Difficulty in accessing data
 - Need to write a new program to carry out each new task
- Data isolation
 - Multiple files and formats
- Integrity problems
 - Integrity constraints (e.g., account balance > 0) become “buried” in program code rather than being stated explicitly
 - Hard to add new constraints or change existing ones

Purpose of Database Systems (Cont.)

- Atomicity of updates
 - Failures may leave database in an inconsistent state with partial updates carried out
 - Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent access by multiple users
 - Concurrent access needed for performance
 - Uncontrolled concurrent accesses can lead to inconsistencies
 - Ex: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time
- Security problems
 - Hard to provide user access to some, but not all, data

Database systems offer solutions to all the above problems. In module I, we explored how users interact with the (some) of the functions through DDL and DML. In module II, we will explore *how* DBMS implement the capabilities “under the covers.”

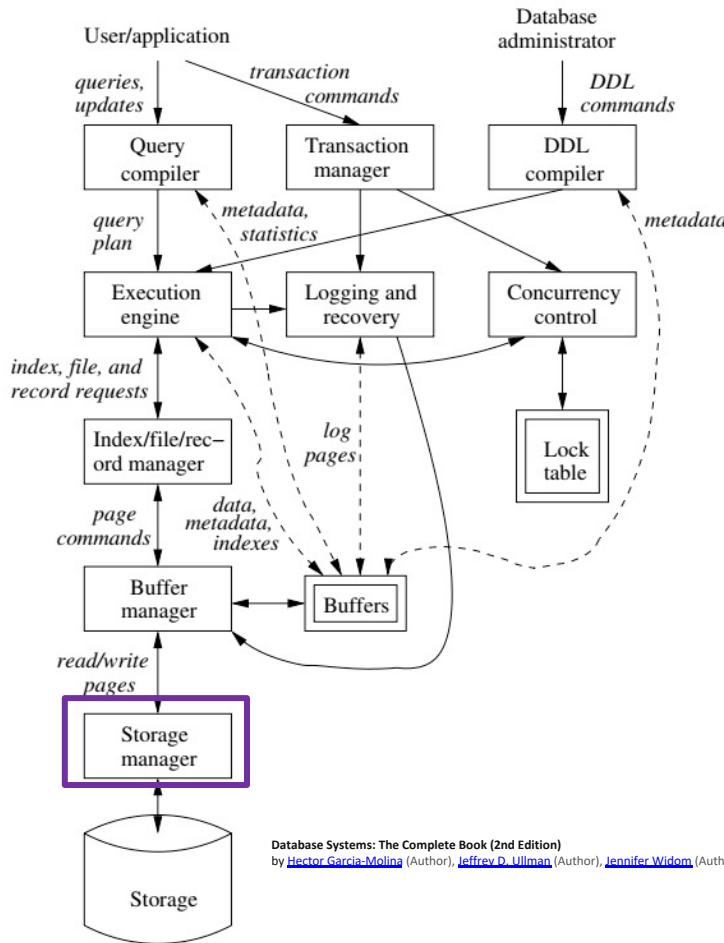
DBMS Arch.



Data Management

Today

- Load/save things quickly.



Disks

Input/Output (IO)

Disks as Far as the Eye can See

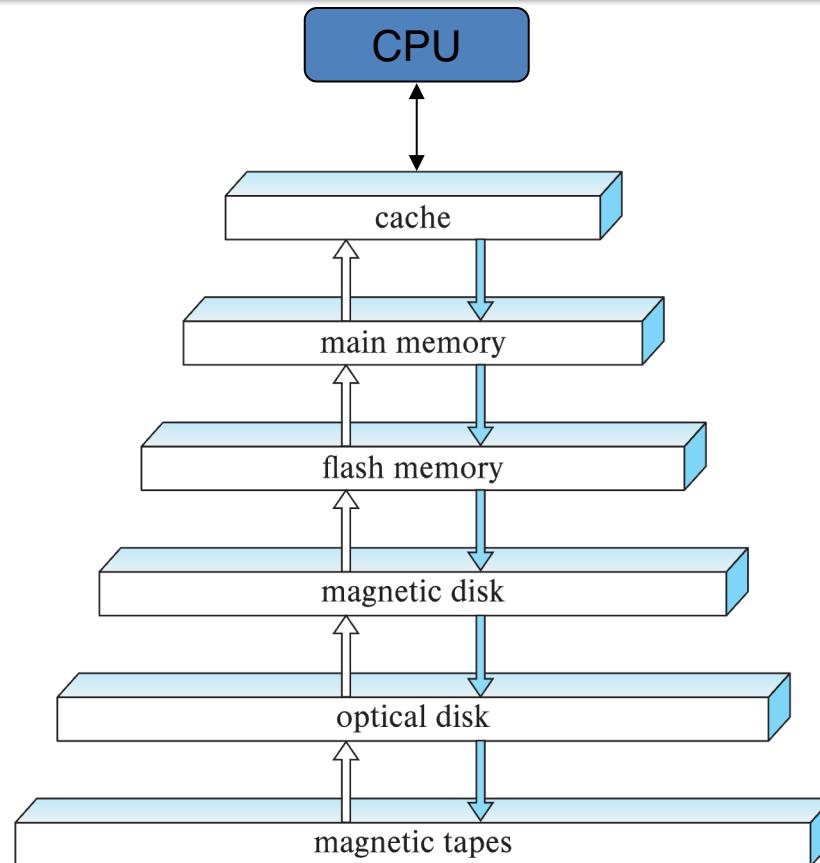


Classification of Physical Storage Media

- Can differentiate storage into:
 - **volatile storage:** loses contents when power is switched off
 - **non-volatile storage:**
 - Contents persist even when power is switched off.
 - Includes secondary and tertiary storage, as well as battery-backed up main-memory.
- Factors affecting choice of storage media include
 - Speed with which data can be accessed
 - Cost per unit of data
 - Reliability

From: Database System Concepts, 7th Ed.

Storage Hierarchy



From: Database System Concepts, 7th Ed.

Memory Hierarchy

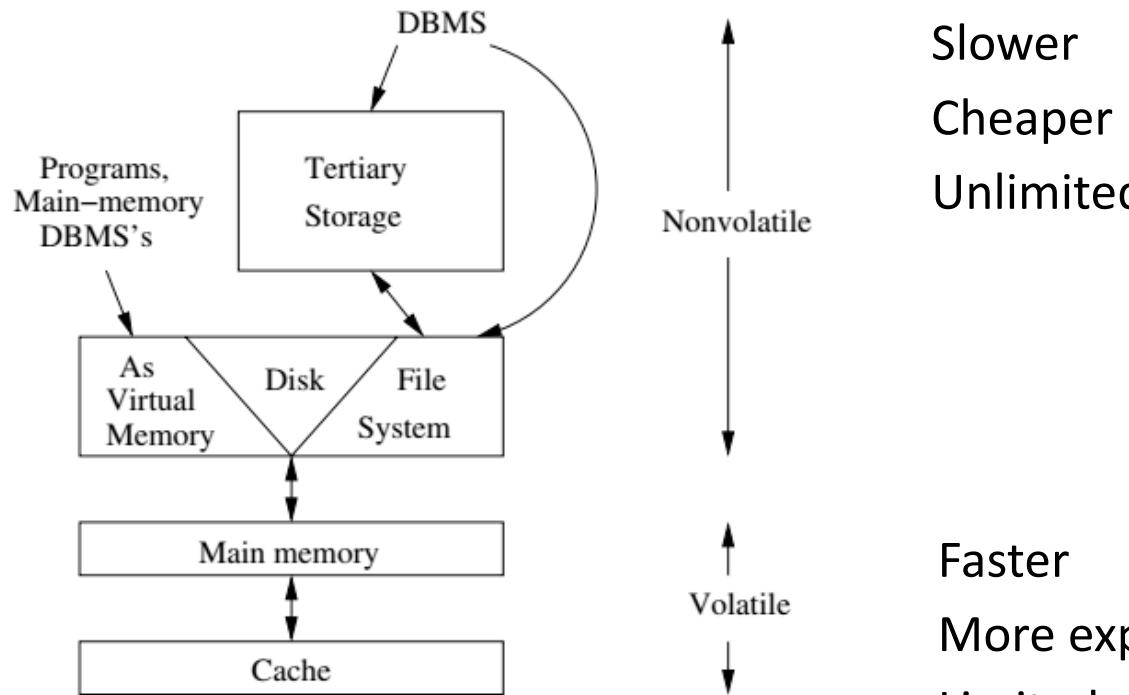


Figure 13.1: The memory hierarchy

Slower
Cheaper
Unlimited

Faster
More expensive
Limited

From: Database System Concepts, 7th Ed.

Memory Hierarchy (Very Old Numbers – Still Directionally Valid)

Storage Technology

Price, Performance & Capacity

Technologies	Capacity (GB)	Latency (microS)	IOPs	Cost/IOPS (\$)	Cost/GB (\$)
Cloud Storage	Unlimited	60,000	20	17c/GB	0.15/month
Capacity HDDs	2,500	12,000	250	1.67	0.15
Performance HDDs	300	7,000	500	1.52	1.30
SSDs (write)	64	300	5000	0.20	13
SSDs (read only)	64	45	30,000	0.03	13
DRAM	8	0.005	500,000	0.001	52

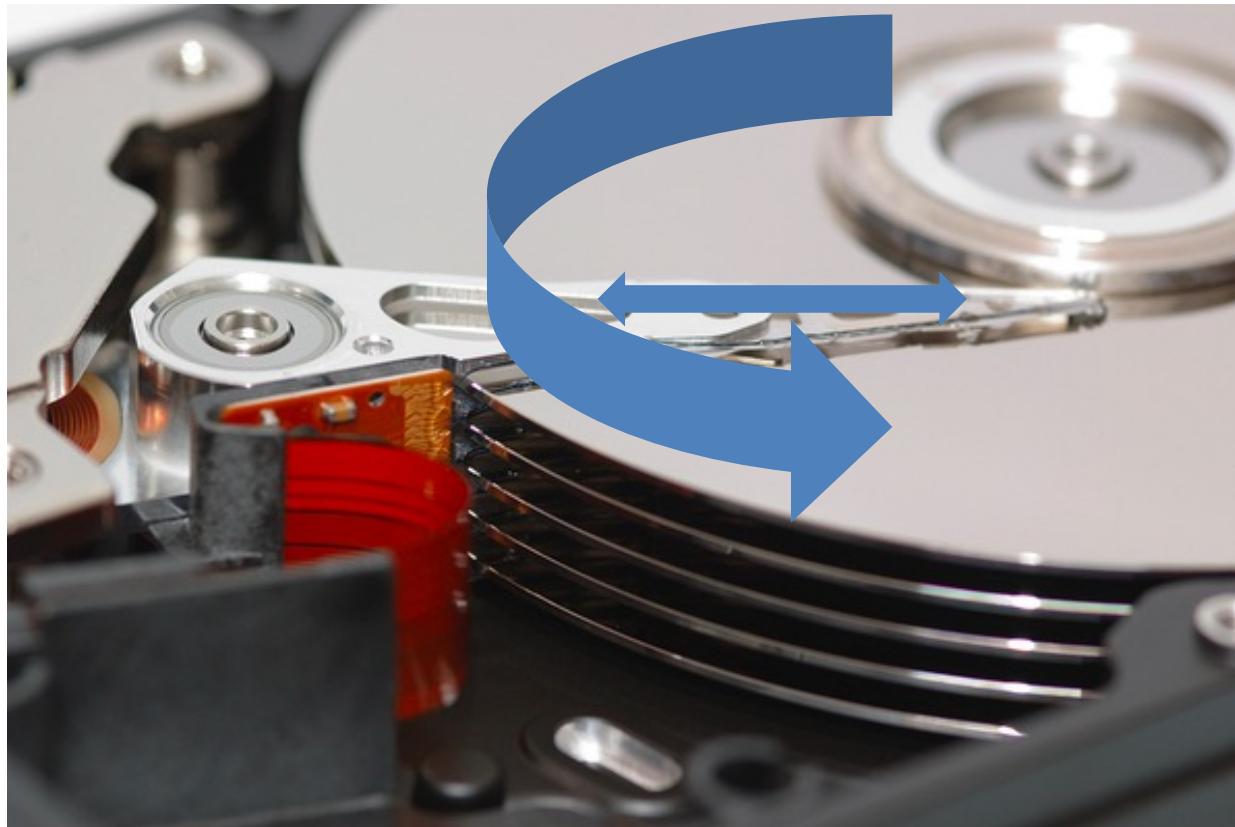
- These numbers are ancient.
- Looking for more modern numbers.
- But, does give an idea of
 - Price
 - Performance
- The general observation is that
 - Performance goes up 10X/level.
 - Price goes up 10x per level.
- Note: One major change is improved price performance of SSD relative to HDD for large data.

Storage Hierarchy (Cont.)

- **primary storage:** Fastest media but volatile (cache, main memory).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
 - Also called **on-line storage**
 - E.g., flash memory, magnetic disks
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
 - also called **off-line storage** and used for **archival storage**
 - e.g., magnetic tape, optical storage
 - Magnetic tape
 - Sequential access, 1 to 12 TB capacity
 - A few drives with many tapes
 - Juke boxes with petabytes (1000's of TB) of storage

From: Database System Concepts, 7th Ed.

Hard Disk Drive



Disk Configuration

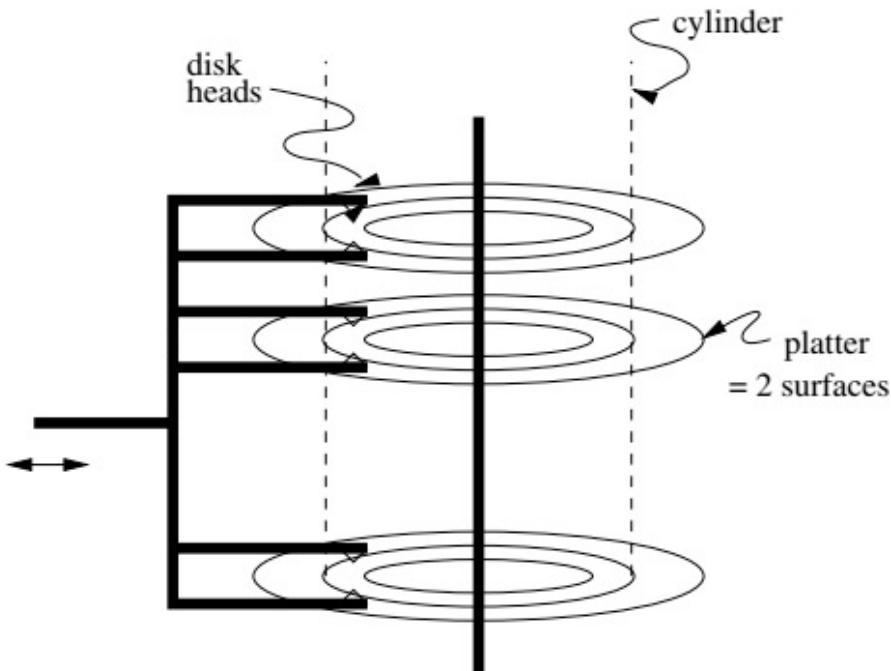


Figure 13.2: A typical disk

Components of disk I/O delay

Seek: Move head to cylinder/track.

Rotation: Wait for sector to get under head.

Transfer: Move data from disk to memory.

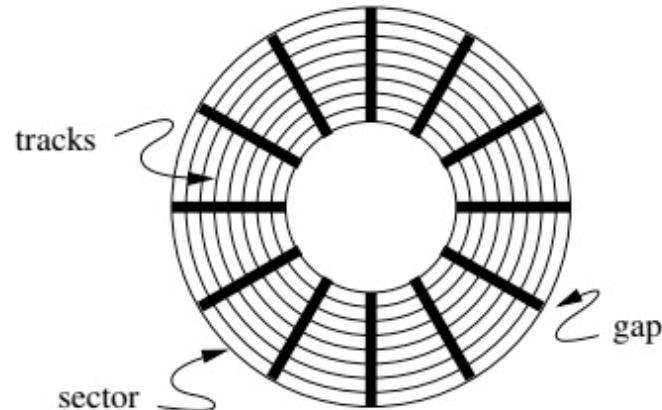
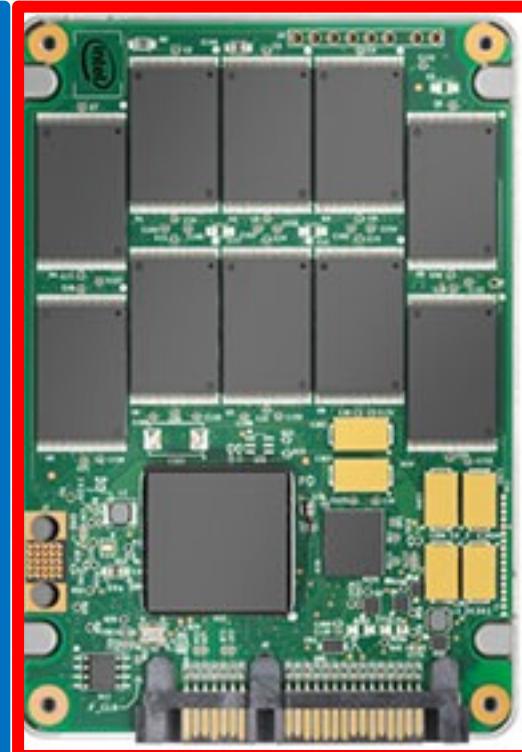


Figure 13.3: Top view of a disk surface

Database Systems: The Complete Book (2nd Edition)
by [Hector Garcia-Molina](#) (Author), [Jeffrey D. Ullman](#) (Author), [Jennifer Widom](#) (Author)

Hard Disk versus Solid State Disk

Hard
Disk
Drive



Solid
State
Drive

Flash Storage

- NOR flash vs NAND flash
- NAND flash
 - used widely for storage, cheaper than NOR flash
 - requires page-at-a-time read (page: 512 bytes to 4 KB)
 - 20 to 100 microseconds for a page read
 - Not much difference between sequential and random read
 - Page can only be written once
 - Must be erased to allow rewrite
- Solid state disks
 - Use standard block-oriented disk interfaces, but store data on multiple flash storage devices internally
 - Transfer rate of up to 500 MB/sec using SATA, and up to 3 GB/sec using NVMe PCIe

From: Database System Concepts, 7th Ed.

Despite the radically different implementation, it has a disk oriented API.

Logical Block Addressing

- Concept:
 - The *unit of transfer* from a “disk” to the computers memory is a “block.
Blocks are usually relatively large, e.g. 16 KB, 32 KB,
 - A program that reads or write a single byte, requires the database engine (or file system) to read/write the entire block.
- The address of a block in the entire space of blocks is:
 - (Device ID, Block ID)
 - Block ID is simple 0, 1, 2,
- The disk controller and disk implementation translate the *logical block address* into the *physical address of blocks*.
- The physical address changes over time for various reasons, e.g. performance optimization, internal HW failure, etc.

Logical/Physical Block Addressing

The mapping from LBA to physical block address can change over time.

- Internal HW failure.
- SSD writes in a funny way.
 - You have to erase before writing.
 - So, the SSD (for performance)
 - Writes to an empty block.
 - Erase the original block.
- Performance optimization on HDD
 - Based on block access patterns.
 - Place blocks on cylinder/sector/head in a way to minimize:
 - Seek
 - Rotate

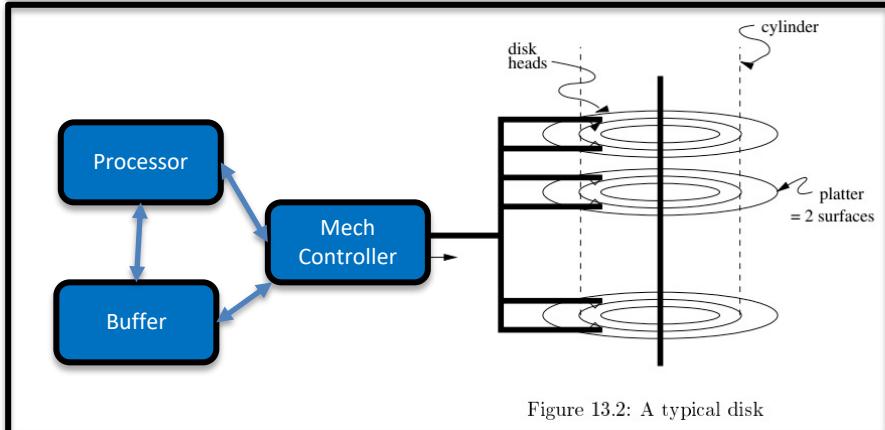
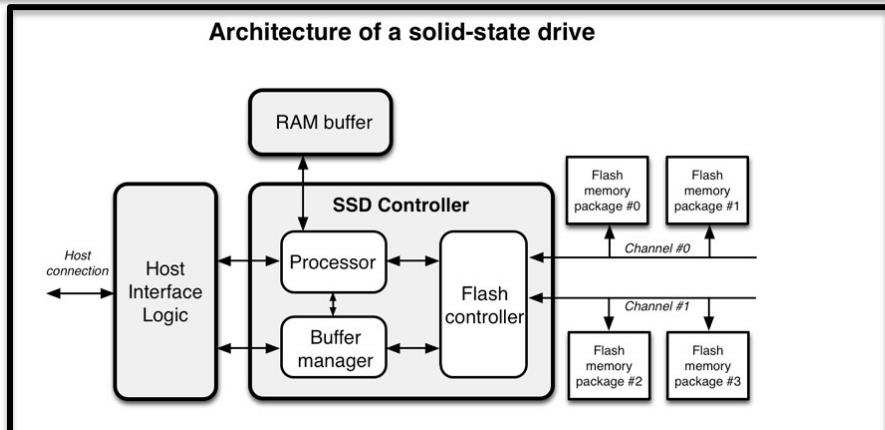
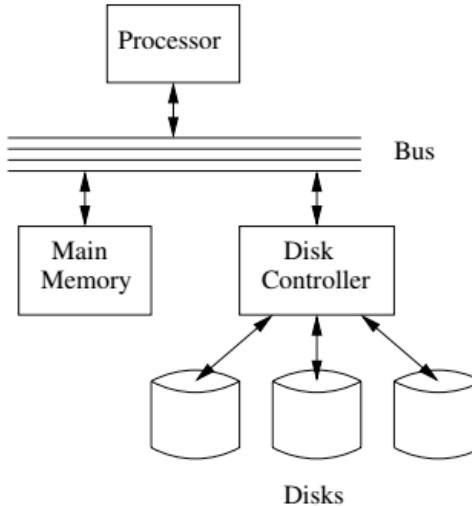


Figure 13.2: A typical disk

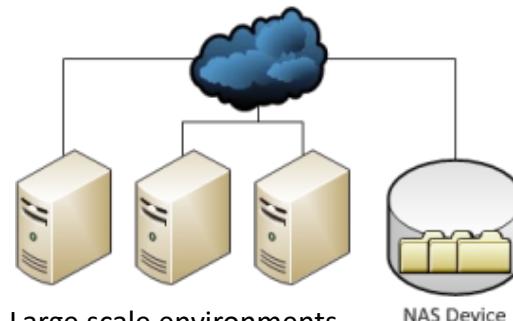
I/O Architecture



How we normally think of disks and I/O.

Network Attached Storage

- Shared storage over shared network
- File system
- Easier management

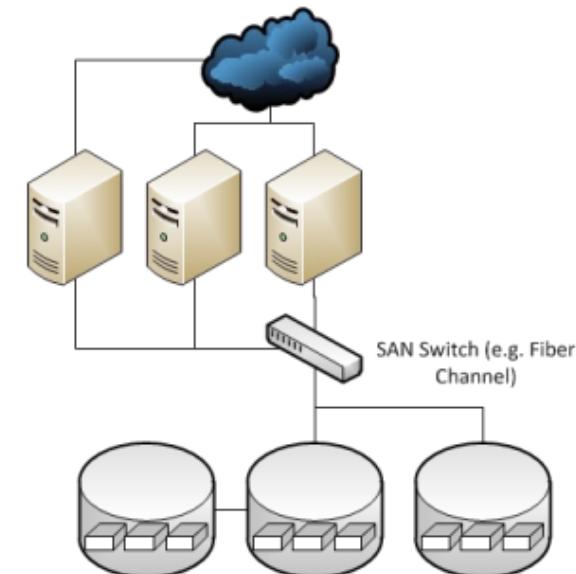


Large scale environments

- The bus-controller connection is over some kind of network.
- The disk controller is at the disks, and basically a “computer” with SW.
- Network is either
 - Standard communication network, or
 - Highly optimized I/O network.

Storage Area Network

- Shared storage over dedicated network
- Raw storage
- Fast, but costly



Magnetic Disks

- **Read-write head**
- Surface of platter divided into circular **tracks**
 - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into **sectors**.
 - A sector is the smallest unit of data that can be read or written.
 - Sector size typically 512 bytes
 - Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)
- To read/write a sector
 - disk arm swings to position head on right track
 - platter spins continually; data is read/written as sector passes under head
- Head-disk assemblies
 - multiple disk platters on a single spindle (1 to 5 usually)
 - one head per platter, mounted on a common arm.
- **Cylinder i** consists of i^{th} track of all the platters

From: Database System Concepts, 7th Ed.

Magnetic Disks (Cont.)

- **Disk controller** – interfaces between the computer system and the disk drive hardware.
 - accepts high-level commands to read or write a sector
 - initiates actions such as moving the disk arm to the right track and actually reading or writing the data
 - Computes and attaches **checksums** to each sector to verify that data is read back correctly
 - If data is corrupted, with very high probability stored checksum won't match recomputed checksum
 - Ensures successful writing by reading back sector after writing it
 - Performs **remapping of bad sectors**

From: Database System Concepts, 7th Ed.

Performance Measures of Disks

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
 - **seek time** – time it takes to reposition the arm over the correct track.
 - Average seek time is 1/2 the worst case seek time.
 - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
 - 4 to 10 milliseconds on typical disks
 - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
 - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
 - Average latency is 1/2 of the above latency.
 - Overall latency is 5 to 20 msec depending on disk model
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
 - 25 to 200 MB per second max rate, lower for inner tracks

From: Database System Concepts, 7th Ed.

Performance Measures (Cont.)

- **Disk block** is a logical unit for storage allocation and retrieval
 - 4 to 16 kilobytes typically
 - Smaller blocks: more transfers from disk
 - Larger blocks: more space wasted due to partially filled blocks
- **Sequential access pattern**
 - Successive requests are for successive disk blocks
 - Disk seek required only for first block
- **Random access pattern**
 - Successive requests are for blocks that can be anywhere on disk
 - Each access requires a seek
 - Transfer rates are low since a lot of time is wasted in seeks
- **I/O operations per second (IOPS)**
 - Number of random block reads that a disk can support per second
 - 50 to 200 IOPS on current generation magnetic disks

From: Database System Concepts, 7th Ed.

Performance Measures (Cont.)

- **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure.
 - Typically 3 to 5 years
 - Probability of failure of new disks is quite low, corresponding to a “theoretical MTTF” of 500,000 to 1,200,000 hours for a new disk
 - E.g., an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
 - MTTF decreases as disk ages

From: Database System Concepts, 7th Ed.

Logical Block Addressing

(https://gerardnico.com/wiki/data_storage/lba)

3 - The LBA scheme

LBA	C	H	S
0	0	0	0
1	0	0	1
2	0	0	2
3	0	0	3
4	0	0	4
5	0	0	5
6	0	0	6
7	0	0	7
8	0	0	8
9	0	0	9
10	0	1	0
11	0	1	1
12	0	1	2
13	0	1	3
14	0	1	4
15	0	1	5
16	0	1	6
17	0	1	7
18	0	1	8
19	0	1	9
Cylinder 0			

LBA	C	H	S
20	1	0	0
21	1	0	1
22	1	0	2
23	1	0	3
24	1	0	4
25	1	0	5
26	1	0	6
27	1	0	7
28	1	0	8
29	1	0	9
30	1	1	0
31	1	1	1
32	1	1	2
33	1	1	3
34	1	1	4
35	1	1	5
36	1	1	6
37	1	1	7
38	1	1	8
39	1	1	9
Cylinder 1			

lowing formula:

tor number

Redundant Array of Independent Disks (RAID)



“RAID (redundant array of independent disks) is a data [storage virtualization](#) technology that combines multiple physical [disk drive](#) components into a single logical unit for the purposes of [data redundancy](#), performance improvement, or both. (...)

[RAID 0](#) consists of [striping](#), without [mirroring](#) or [parity](#). (...)

[RAID 1](#) consists of data mirroring, without parity or striping. (...)

[RAID 2](#) consists of bit-level striping with dedicated [Hamming-code](#) parity. (...)

[RAID 3](#) consists of byte-level striping with dedicated parity. (...)

[RAID 4](#) consists of block-level striping with dedicated parity. (...)

[RAID 5](#) consists of block-level striping with distributed parity. (...)

[RAID 6](#) consists of block-level striping with double distributed parity. (...)

Nested RAID

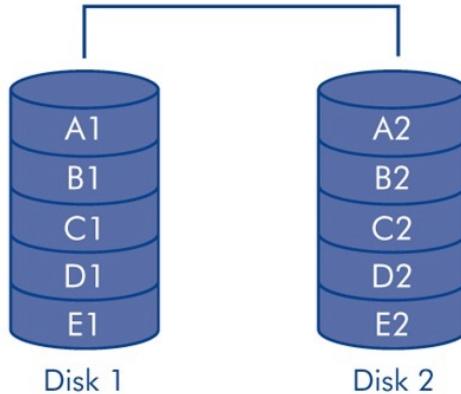
- RAID 0+1: creates two stripes and mirrors them. (...)
- RAID 1+0: creates a striped set from a series of mirrored drives. (...)
- **[JBOD RAID N+N](#)**: With JBOD (*just a bunch of disks*), (...)"

RAID-0 and RAID-1

Two physical disks make
one single, logical **fast** disk



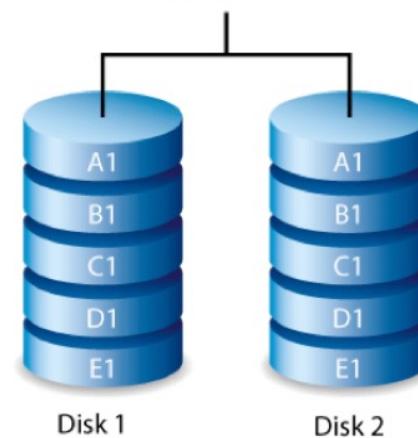
RAID 0



Two physical disks make
one single, logical **reliable** disk

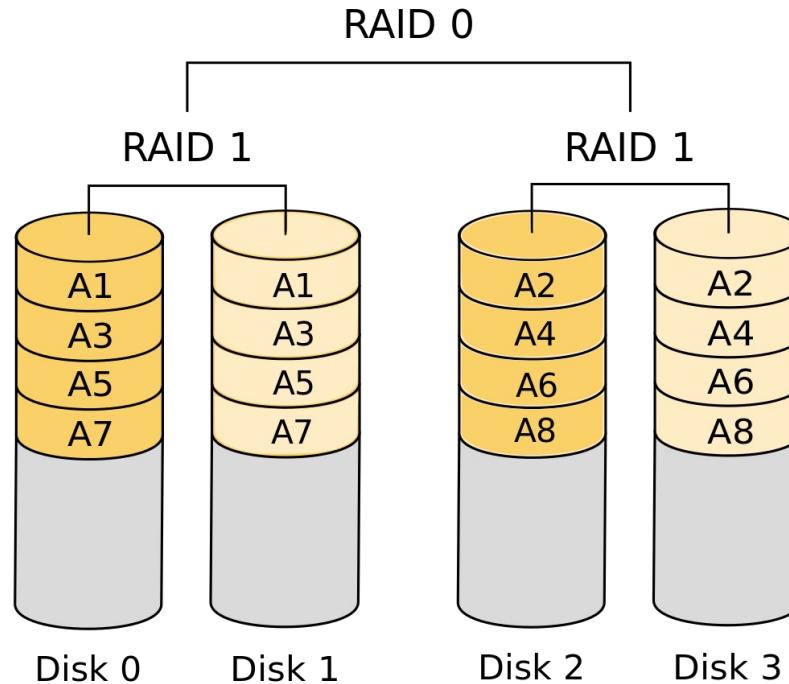


RAID 1



Mixed RAID Modes

RAID 1+0



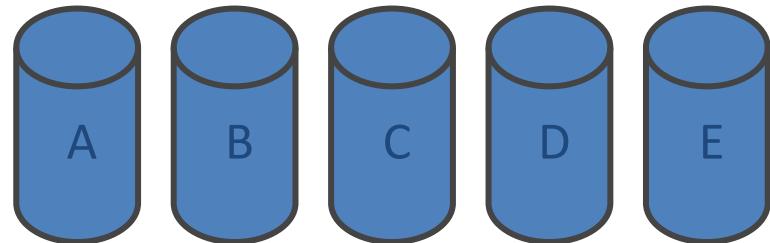
Stripe
And
Mirror

RAID-5

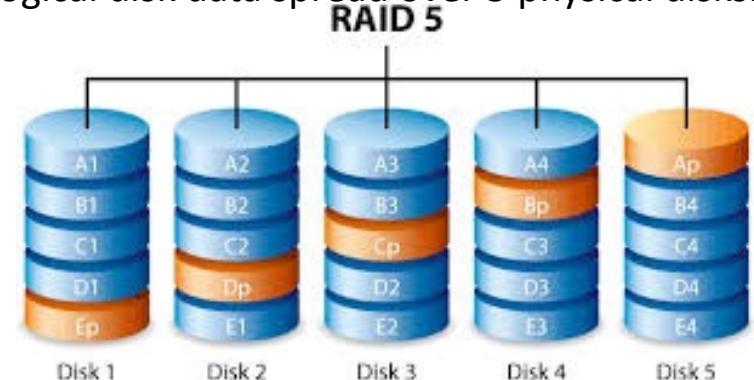
- Improved performance through parallelism
 - Rotation/seek
 - Transfer
- Availability uses *parity blocks*
 - Suppose I have 4 different data blocks on the logical drive A: A1, A2, A3, A4.
 - Parity function: $A_p = P(A_1, A_2, A_3, A_4)$
 - Recovery function: $A_2 = R(A_p, A_1, A_3, A_4)$
- During normal operations:
 - Read processing simply retrieves block.
 - Write processing of A2 updates A2 and Ap
- If an individual disk fails, the RAID
 - Read
 - Continues to function for reads on non-missing blocks.
 - Implements read on missing block by recalculating value.
 - Write
 - Updates block and parity block for non-missing blocks.
 - Computes missing block, and calculates parity based on old and new value.
 - Over time
 - “Hot Swap” the failed disk.
 - Rebuild the missing data from values and parity.



Is actually 5 smaller “logical” disks.

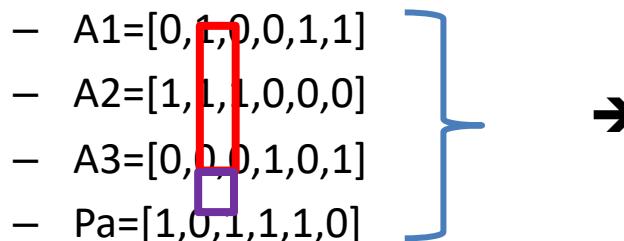


Logical disk data spread over 5 physical disks.



Very Simple Parity Example

- Even-Odd Parity
 - $b[i]$ is an array of bits (0 or 1)
 - $P(b[i]) =$
 - 0 if an even number of bits = 1. $\{P([0,1,1,0,1,1])=0\}$
 - 1 if an odd number of bits = 1. $\{P(0,0,1,0,1,1)=1\}$
 - Given an array with one missing bit and the parity bit, I can re-compute the missing bit.
 - Case 1: $[0,?,1,0,1,1]$ has $P=0$. There must be an EVEN number of ones and $?=1$.
 - Case 2: $[0,?,1,0,1,1]$ has $P=1$. There must be an ODD number of ones and $?=0$.
- Block Parity applies this to a set of blocks bitwise



If I am missing a block and have the parity block, I can re-compute the missing block bitwise from remaining blocks and parity block.

NoSQL

Concepts

Overview (I) (<https://en.wikipedia.org/wiki/NoSQL>)

A **NoSQL** (originally referring to "non SQL" or "non relational")^[1] database provides a mechanism for storage and retrieval of data that is modeled in **means other than the tabular relations used in relational databases**. Such databases have existed since the late 1960s, but did not obtain the "NoSQL" moniker until a surge of popularity in the early twenty-first century,^[2] triggered by the needs of Web 2.0 companies such as Facebook, Google, and Amazon.com.^{[3][4][5]} NoSQL databases are increasingly used in big data and real-time web applications.^[6] NoSQL systems are also sometimes called "**Not only SQL**" to emphasize that they may support SQL-like query languages.^{[7][8]}

Motivations for this approach include: simplicity of design, simpler "horizontal scaling" to clusters of machines (which is a problem for relational databases),^[2] and finer control over availability. The data structures used by NoSQL databases (e.g. key-value, wide column, graph, or document) are different from those used by default in relational databases, making **some operations faster in NoSQL**. The particular suitability of a given NoSQL database depends on the problem it must solve. Sometimes the **data structures used by NoSQL databases are also viewed as "more flexible"** than relational database tables.^[9]

Overview (I) (<https://en.wikipedia.org/wiki/NoSQL>)

Many NoSQL stores compromise [consistency](#) (in the sense of the [CAP theorem](#)) in favor of availability, partition tolerance, and speed. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages (instead of SQL, for instance the lack of ability to perform ad-hoc joins across tables), lack of standardized interfaces, and huge previous investments in existing relational databases.^[10] Most NoSQL stores lack true [ACID](#) transactions,

Instead, most NoSQL databases offer a concept of "eventual consistency" in which database changes are propagated to all nodes "eventually" (typically within milliseconds) so queries for data might not return updated data immediately or might result in reading data that is not accurate, a problem known as stale reads.^[11] Additionally, some NoSQL systems may exhibit lost writes and other forms of [data loss](#).^[12] Fortunately, some NoSQL systems provide concepts such as [write-ahead logging](#) to avoid data loss.^[13] For [distributed transaction processing](#) across multiple databases, data consistency is an even bigger challenge that is difficult for both NoSQL and relational databases. Even current relational databases "do not allow referential integrity constraints to span databases."^[14]

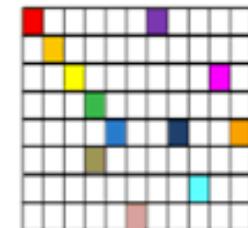
Simplistic Classification

(<https://medium.com/swlh/4-types-of-nosql-databases-d88ad21f7d3b>)

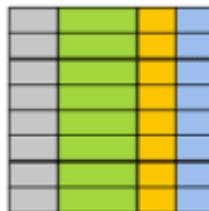
Relational is the foundational model.

We covered graphs and examples.

Column-Family

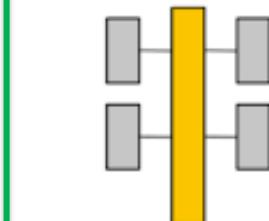


Relational



SQL Database

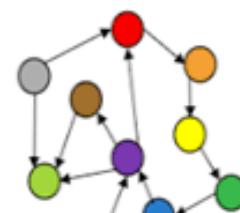
Analytical (OLAP)



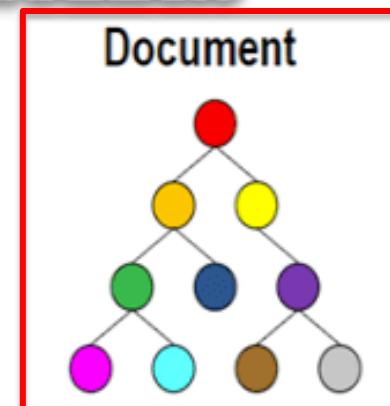
We will see OLAP in a future lecture.

NoSQL Database

Graph

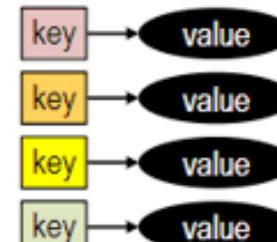


Document



Subject of this lecture and part of HW4

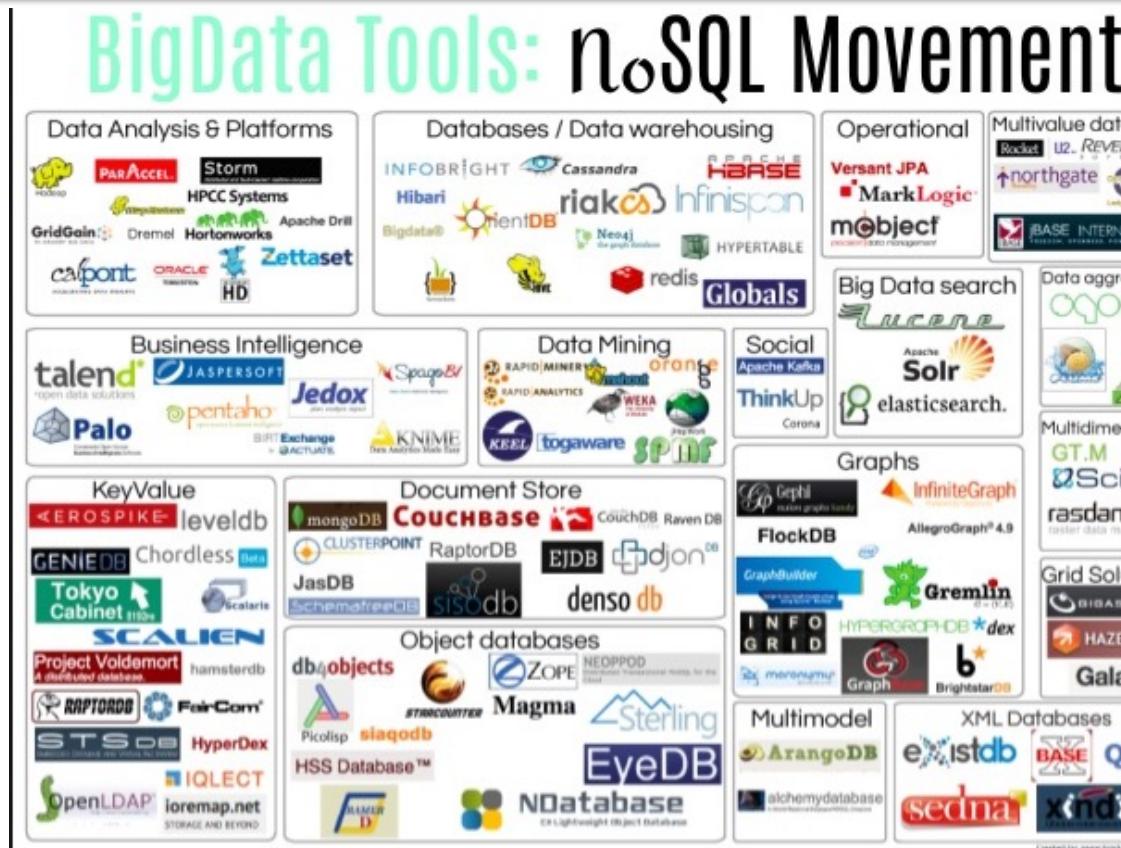
Key-Value



One Taxonomy

Document Database	Graph Databases
   	  The Distributed Graph Database
Wide Column Stores	Key-Value Databases
   AEROSPIKE	   HBASE  

Another Taxonomy



Use Cases

Motivations

- Massive write performance.
- Fast key value look ups.
- Flexible schema and data types.
- No single point of failure.
- Fast prototyping and development.
- Out of the box scalability.
- Easy maintenance.

What is wrong with SQL/Relational?

- Nothing. One size fits all? Not really.
- Impedance mismatch. – Object Relational Mapping doesn't work quite well.
- Rigid schema design.
- Harder to scale.
- Replication.
- Joins across multiple nodes? Hard.
- How does RDMS handle data growth? Hard.
- Need for a DBA.
- Many programmers are already familiar with it.
- Transactions and ACID make development easy.
- Lots of tools to use.

Documents

Example Document – An Order

FOOD ORDER FORM TEMPLATE

Company Name
123 Main Street
Hamilton, OH 44416
(321) 456-7890
Email Address
Point of Contact
web address

YOUR LOGO

ORDER FORM

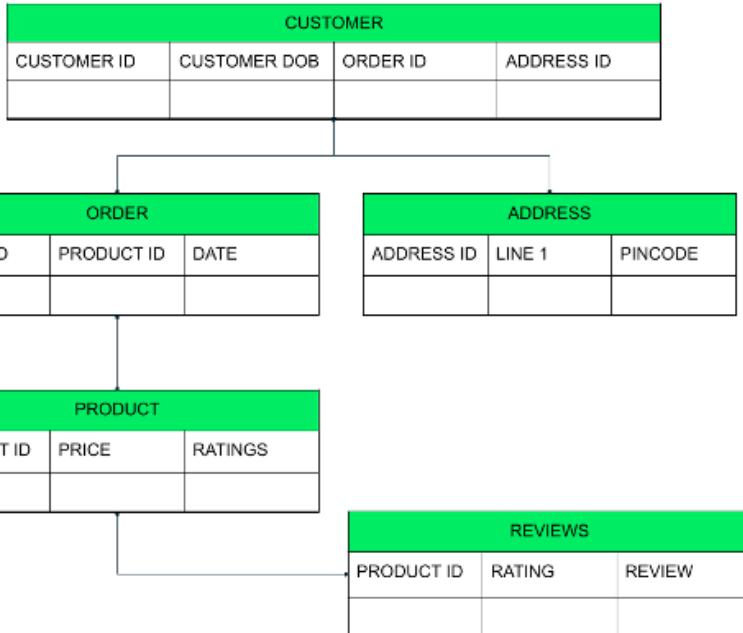
CUSTOMER		ORDER NO.	ORDER DATE
ATTN: Name / Dept		DATE NEEDED	TIME NEEDED
Company Name		ORDER RECEIVED BY	
123 Main Street			
Hamilton, OH 44416			
(321) 456-7890			
Email Address			
DESCRIPTION			
		UNIT PRICE	QUANTITY
		\$ -	
		\$ -	
		\$ -	
		\$ -	
		\$ -	
		\$ -	
		\$ -	
		\$ -	
		\$ -	
FINANCIALS			
enter percentage		TAX RATE	0.000%
enter initial pymt amount		TOTAL TAX	\$ -
		DELIVERY	\$ -
		GRAND TOTAL	\$ -
		LESS PAYMENT	\$ -

THANK YOU!

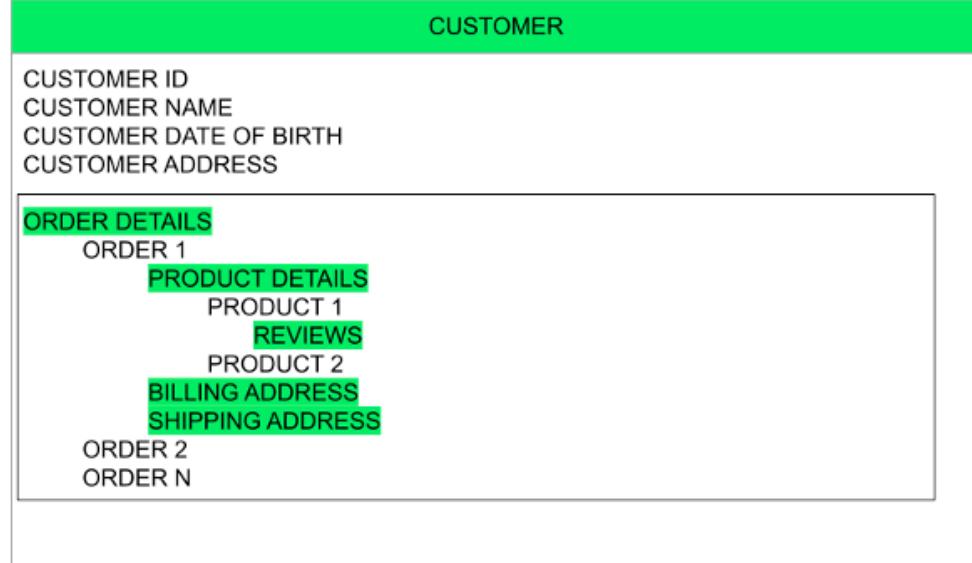
- There are 5 entity types on the form:
 - A “copy of” customer information.
 - Links (via productCode) to products
 - Order
 - OrderDetails
 - Comments
- But OrderDetails and Comments are somehow different from the others.
 - These are arrays of objects
 - That are sort of “inside” the order.
- These are weak entities, but relational does not always handle these well.

Relational vs Document

Relational



Relational



The key difference?

- Relational DB attributes are atomic.
- Document DB attributes may be *multi-valued* and *complex*.

UI Interface and Logical Data Model

{

Columbia University

CUSTOMER NUMBER

10001

CONTACT NAME

Lord Voldemort

CONTACT NUMBER

+1 212-555-6666

Customer Address (THIS WAS INSANELY PAINFUL TO LAYOUT)

STREET 1

520 W 120th St.

STREET 2

4th Floor

CITY

New York

STATE

NY

COUNTRY

US

POSTAL CODE

10027

I nearly went "postal" doing the previous layout and almost failed all of you!

NUMBER	STATUS	ORDER DATE	REQUIRED DATE	SHIPPED DATE	DETAILS
21	Shipped	2022-01-01	2022-02-01	2022-01-28	<button>Expand</button>

LINE NUMBER PRODUCT CODE QUANTITY PRICE EACH

0	P0	0	0
1	P1	2	3
2	P2	4	6

NUMBER	STATUS	ORDER DATE	REQUIRED DATE	SHIPPED DATE	DETAILS
22	Pending	2022-03-01	2022-04-01	0000000	<button>Expand</button>

customerName: "Columbia University",

contact: {

name: "Lord Voldemort",

phone: "+1 212-555-6666"

},

address: {

line1: "520 W 120th St."

line2: "Floor 4",

... ...

}

orders: [

{

number: ...,

... ...

details: [

... ...

]

}

}

Switch to Notebook

MongoDB

MongoDB Concepts

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key <code>_id</code> provided by MongoDB itself)
Database Server, Client, Tools, Packages	
mysqld/Oracle	<code>mongod</code>
mysql/sqlplus	<code>mongo</code>
DataGrip	Compass
pymysql	<code>pymongo</code>

Core Operations

Basic Operations:

- Create database
- Create collection
- Create-Retrieve-Update-Delete (CRUD):
 - Create: insert()
 - Retrieve:
 - find()
 - find_one()
 - Update: update()
 - Delete: remove()

More Advanced Concepts:

- Limit
- Sort
- Aggregation Pipelines
 - Merge
 - Union
 - Lookup
 - Match
 - Merge
 - Sample
 -

We will just cover the basics for now and may cover more things in HW or other lectures.

find()

- Note:
 - MongoDB uses a more `pymysql` approach, e.g. an API, than pure declarative languages like SQL.
 - The parameters for `find()` are where the declarative language appears.
- The basic forms of `find()` and `find_one()` have two parameters:
 - *filter expression*
 - *Project expression*
- You can use the Compass tool and screen captures for some HW and exam answers.
- What if I want the answer in a Jupyter Notebook?

The screenshot shows the MongoDB Compass interface with the following details:

- Collection:** GOT.seasons
- Documents:** 8 (TOTAL SIZE 1.0MB, AVG. SIZE 130.2KB)
- Indexes:** 1 (TOTAL SIZE 20.0KB, AVG. SIZE 20.0KB)
- Filter:** `{"episodes.scenes.location": "The Dothraki Sea"}`
- Project:** `{ season: 1, "episodes.episodeNum":1, "episodes.episodeLink": 1, "episodes.episodeTitle": 1 }`
- Sort:** `{ field: -1 }`
- Collation:** `{ locale: 'simple' }`
- Max Time MS:** 60000
- Skip:** 0
- Limit:** 0
- Buttons:** FIND, RESET, ...
- Display:** Displaying documents 1 - 3 of 3
- Document Structure:** The interface shows three document results. The first document has an _id of `"60577e50c68b67110968b6d1"`, a season of `"1"`, and an episodes array containing 10 objects. The second document has an _id of `"60577e50c68b67110968b6d5"`, a season of `"5"`, and an episodes array. The third document has an _id of `"60577e50c68b67110968b6d6"`, a season of `"5"`, and an episodes array.

Generate Code

The screenshot shows the MongoDB Compass interface. On the left, the sidebar lists databases (Local, HOST localhost:27017, CLUSTER Standalone, EDITION MongoDB 4.2.6 Community) and collections (GOT, characters, seasons, admin, config, db, fantasy_baseball, local). In the center, the 'GOT.seasons' collection is selected, showing two documents. A modal window titled 'Export Query To Language' is open, containing a query builder for a MongoDB query. The 'My Query:' field contains the following code:

```
1 {  
2   "episodes.scenes.location": "The Dothraki Sea"  
3 }
```

The 'Export Query To:' dropdown is set to 'PYTHON 3'. The generated Python code is:

```
1 # Requires the PyMongo package.  
2 # https://api.mongodb.com/python/current  
3  
4 client = MongoClient('mongodb://localhost:27017/?  
5 filter={  
6   'episodes.scenes.location': 'The Dothraki Sea'  
7 }  
8 project={  
9   'season': 1,  
10  'episodes.episodeNum': 1,  
11  'episodes.episodeLink': 1,  
12  'episodes.episodeTitle': 1  
13 }  
14  
15 result = client['GOT']['seasons'].find(  
16   filter=filter,
```

Below the code, there are two checkboxes: 'Include Import Statements' (unchecked) and 'Include Driver Syntax' (checked). At the bottom right of the modal is a 'CLOSE' button.

- Choose Export to Language.
- Copy into the notebook.
- The export has an option to include all the connection setup, choosing DB,
- Switch to Notebook

Result is not Quite You Expect

GOT.seasons

DOCUMENTS 8 TOTAL SIZE 1.0MB AVG. SIZE 130.2KB INDEXES 1 TOTAL SIZE 20.0KB AVG. SIZE 20.0KB

Documents Aggregations Schema Explain Plan Indexes Validation

FILTER: {"episodes.scenes.location": "The Dothraki Sea"}
PROJECT: { season: 1, "episodes.episodeNum":1, "episodes.episodeLink": 1, "episodes.episodeTitle": 1}
SORT: { field: -1 }
COLLATION: { locale: 'simple' }

FIND RESET ...
MAX TIME MS: 60000
SKIP: 0 LIMIT: 0

VIEW Refresh

Displaying documents 1 - 3 of 3 < > C REFRESH

```
_id:ObjectId("60577e50c68b67110968b6d1")
season:"1"
episodes:Array
  ▾ 0:Object
    episodeNum: 1
    episodeTitle: "Winter Is Coming"
    episodeLink: "/title/tt1480055/"
  ▾ 1:Object
  ▾ 2:Object
  ▾ 3:Object
    episodeNum: 4
    episodeTitle: "Cripples, Bastards, and Broken Things"
    episodeLink: "/title/tt1829963/"
  ▾ 4:Object
  ▾ 5:Object
  ▾ 6:Object
  ▾ 7:Object
  ▾ 8:Object
  ▾ 9:Object

_id:ObjectId("60577e50c68b67110968b6d5")
season:"5"
episodes:Array

_id:ObjectId("60577e50c68b67110968b6d6")
season:"6"
episodes:Array
```

- The query returns documents that match.
 - The document is “Large” and has seasons and episodes and seasons.
 - If you do a \$project requesting episodes/episode content,
 - You get all episodes in the documents that match.
 - Not just the episodes with the scene/location.
 - Projecting array elements from arrays whose elements are arrays is complex and baffling.
- You also get back something (a cursor) that is iterable.

Result is not Quite You Expect

The screenshot shows the MongoDB Compass interface with a query results grid and the MongoDB shell command history.

Query Results Grid:

_id	season	episodes
<code>_id: ObjectId("60577e50c68b67110968b6d1")</code>	"5"	[{"episodeNum": 1, "episodeTitle": "Winter Is Coming", "episodeLink": "/title/tt1480055"}, {"episodeNum": 2, "episodeTitle": "The Night King", "episodeLink": "/title/tt1480056"}, {"episodeNum": 3, "episodeTitle": "Cripples, Bastards, and Broken Men", "episodeLink": "/title/tt1829963"}, {"episodeNum": 4, "episodeTitle": "The Children", "episodeLink": "/title/tt1480057"}, {"episodeNum": 5, "episodeTitle": "The Mountain and the Viper", "episodeLink": "/title/tt1480058"}, {"episodeNum": 6, "episodeTitle": "The Red and the Black", "episodeLink": "/title/tt1480059"}, {"episodeNum": 7, "episodeTitle": "The Spoils of War", "episodeLink": "/title/tt1480060"}, {"episodeNum": 8, "episodeTitle": "The Mountain Versus the Viper", "episodeLink": "/title/tt1480061"}, {"episodeNum": 9, "episodeTitle": "The Rains of Castamere", "episodeLink": "/title/tt1480062"}]
<code>_id: ObjectId("60577e50c68b67110968b6d5")</code>	"5"	[{"episodeNum": 1, "episodeTitle": "Winter Is Coming", "episodeLink": "/title/tt1480055"}, {"episodeNum": 2, "episodeTitle": "The Night King", "episodeLink": "/title/tt1480056"}, {"episodeNum": 3, "episodeTitle": "Cripples, Bastards, and Broken Men", "episodeLink": "/title/tt1829963"}, {"episodeNum": 4, "episodeTitle": "The Children", "episodeLink": "/title/tt1480057"}, {"episodeNum": 5, "episodeTitle": "The Mountain and the Viper", "episodeLink": "/title/tt1480058"}, {"episodeNum": 6, "episodeTitle": "The Red and the Black", "episodeLink": "/title/tt1480059"}, {"episodeNum": 7, "episodeTitle": "The Spoils of War", "episodeLink": "/title/tt1480060"}, {"episodeNum": 8, "episodeTitle": "The Mountain Versus the Viper", "episodeLink": "/title/tt1480061"}, {"episodeNum": 9, "episodeTitle": "The Rains of Castamere", "episodeLink": "/title/tt1480062"}]
<code>_id: ObjectId("60577e50c68b67110968b6d6")</code>	"6"	[{"episodeNum": 1, "episodeTitle": "Winter Is Coming", "episodeLink": "/title/tt1480055"}, {"episodeNum": 2, "episodeTitle": "The Night King", "episodeLink": "/title/tt1480056"}, {"episodeNum": 3, "episodeTitle": "Cripples, Bastards, and Broken Men", "episodeLink": "/title/tt1829963"}, {"episodeNum": 4, "episodeTitle": "The Children", "episodeLink": "/title/tt1480057"}, {"episodeNum": 5, "episodeTitle": "The Mountain and the Viper", "episodeLink": "/title/tt1480058"}, {"episodeNum": 6, "episodeTitle": "The Red and the Black", "episodeLink": "/title/tt1480059"}, {"episodeNum": 7, "episodeTitle": "The Spoils of War", "episodeLink": "/title/tt1480060"}, {"episodeNum": 8, "episodeTitle": "The Mountain Versus the Viper", "episodeLink": "/title/tt1480061"}, {"episodeNum": 9, "episodeTitle": "The Rains of Castamere", "episodeLink": "/title/tt1480062"}]

MongoDB Shell Command History:

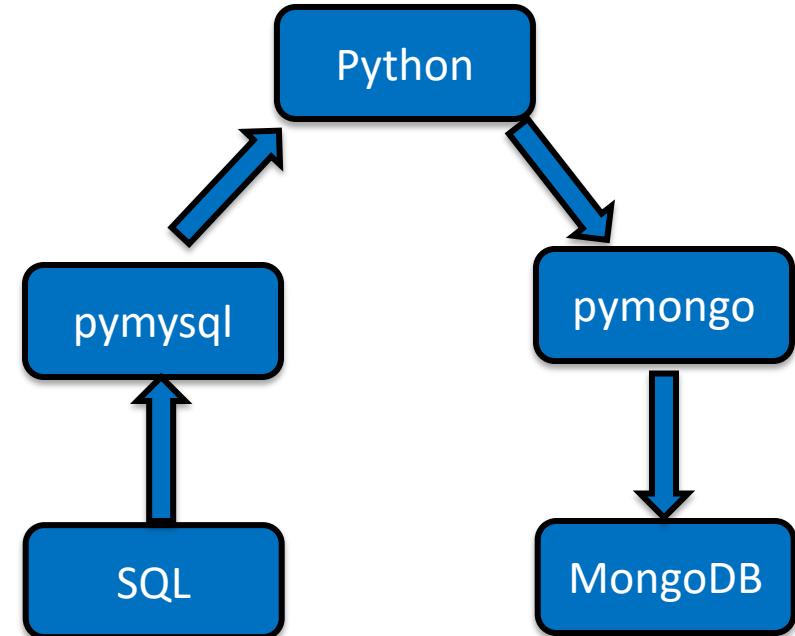
```
db.GOT.seasons.find({ "episodes.scenes.location": "The Dothraki Sea" })  
db.GOT.seasons.project({ season: 1, episodes.episodeNum: 1, episodes.episodeLink: 1, episodes.episodeTitle: 1 })  
db.GOT.seasons.sort({ field: -1 })  
db.GOT.seasons.collation({ locale: 'simple' })
```

- The query returns documents that match.
 - The document is “Large” and has many episodes and seasons.
 - You do a \$project requesting episodes/episode content,
 - You get all episodes in the documents that match.
 - Not just the episodes with the scene/location.
 - Projecting array elements from arrays whose elements are arrays is complex and baffling.
 - You also get back something (a cursor) that is iterable.

More Fun – Data Types

MongoDB supports many datatypes. Some of them are –

- **String** – This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** – This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** – This type is used to store a boolean (true/ false) value.
- **Double** – This type is used to store floating point values.
- **Min/ Max keys** – This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** – This type is used to store arrays or list or multiple values into one key.
- **Timestamp** – timestamp. This can be handy for recording when a document has been modified or added.
- **Object** – This datatype is used for embedded documents.
- **Null** – This type is used to store a Null value.
- **Symbol** – This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date** – This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** – This datatype is used to store the document's ID.
- **Binary data** – This datatype is used to store binary data.
- **Code** – This datatype is used to store JavaScript code into the document.
- **Regular expression** – This datatype is used to store regular expression.



(Some) MongoDB CRUD Operations

- Create:
 - db.collection.insertOne()
 - db.collection.insertMany()
- Retrieve:
 - db.collection.find()
 - db.collection.findOne()
 - db.collection.findOneAndUpdate()
 -
- Update:
 - db.collection.updateOne()
 - db.collection.updateMany()
 - db.collection.replaceOne()
- Delete:
 - db.collection.deleteOne()
 - db.collection.deleteMany()

pymongo maps the camel case to _, e.g.

- findOne()
- find_one()

There are good online tutorials:

- https://www.tutorialspoint.com/python_data_access
- <https://www.tutorialspoint.com/mongodb/index.htm>

(Some) MongoDB Pipeline Operators

<https://www.slideshare.net/mongodb/s01-e04-analytics>

Aggregation operators

- Pipeline and Expression operators

Pipeline	Expression	Arithmetic	Conditional
\$match	\$addToSet	\$add	\$cond
\$sort	\$first	\$divide	\$ifNull
\$limit	\$last	\$mod	
\$skip	\$max	\$multiply	
\$project	\$min	\$subtract	
\$unwind	\$avg		Variables
\$group	\$push		
\$geoNear	\$sum		
\$text			\$let
\$search			\$map

Tip: Other operators for date, time, boolean and string manipulation

MongoDB Checkpoint

- You can see that MongoDB has powerful, sophisticated
 - Operators
 - Expressions
 - Pipelines
- We have only skimmed the surface. There is a lot more:
 - Indexes
 - Replication, Sharding
 - Embedded Map-Reduce support
 -
- We will explore a little more in subsequent lectures, homework,
- You will have to install MongoDB and Compass for HW4 and final exam.