Machine Learning for Data Management Systems

Background

Amol Deshpande
Jan 31, 2023

Outline

- **Examples of Data Management Systems**
- Overview
- Architecture of Database System

Database System

- A DBMS is a software system designed to store, manage, facilitate access to databases
- Provides:
  - Data Definition Language (DDL)
    - For defining and modifying the schemas
  - Data Manipulation Language (DML)
    - For retrieving, modifying, analyzing the data itself
  - Guarantees about correctness in presence of failures and concurrency, data semantics etc. (e.g., ACID guarantees)
- Common use patterns
  - Handling transactions (e.g. ATM Transactions, flight reservations)
  - Archival (storing historical data)
  - Analytics (e.g. identifying trends, **Data Mining**)

1.Relational DBMS: SQL

- **SQL** (sequel): Structured Query Language
- **Data definition (DDL)**
  - **create table** *instructor* (

```
    ID          char(5),
    name        varchar(20),
    dept_name   varchar(20),
    salary      numeric(8,2))
```

- **Data manipulation (DML)**
  - Example: Find the name of the instructor with ID 22222

```
select  name
from    instructor
where   instructor.ID = '22222'
```

1.Some Example Queries

Single-table queries:

select *i.name, i.salary * 2* **as** *double_salary*
from *instructor i*
where *i.salary < 80000* **and** *i.name like '%g_';*

Find the average salary of instructors in the Computer Science

select ***avg(****salary)*
from *instructor*
where *dept_name = 'Comp. Sci';*

Using "joins" to connect information across tables:

select *
from *instructor i, teaches t*
where *i.ID = t.ID;*

1.More Complex Queries

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
        course_id in (select course_id
                      from section
                      where semester = 'Spring' and year= 2010);
```

```
select ID, Salary, rank() over (order by salary desc) as s_rank
    from instructor
    order by s_rank
```

1. Recursion in SQL

- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
    select course_id, prereq_id
    from prereq
  union
    select rec_prereq.course_id, prereq.prereq_id,
    from rec_rereq, prereq
    where rec_prereq.prereq_id = prereq.course_id
  )
select ∗
from rec_prereq;
```

Makes SQL Turing Complete (i.e., you can write any program in SQL)

1. SQL Functions

- Function to count number of instructors in a department

```
create function dept_count (dept_name varchar(20))
     returns integer
         begin
         declare d_count  integer;
              select count (* ) into d_count
              from instructor
              where instructor.dept_name = dept_name
         return d_count;
   end
```

- Can use in queries

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

1. Transactions

insert into students values (…)

update instructor
set salary = salary * 1.03

enrolled = select count(*)
from takes
where (course_info) = (CMSC 424, 201, Fall 2022)
if enrolled < capacity for the room:
insert new student into takes for that course

(Modify prerequisites)
delete (CMSC422, CMSC351)
insert (CMSC422, CMSC320)

(Add a new section for a course for a given room and instructor)
if no section currently in that room:
insert a tuple into "sections" with that room
insert a tuple into "teaches"

(Remove a section)
delete from takes for that section
delete from teaches for that section
delete tuple from section

(Switch the advisor for a student)
delete old tuple with that s_id
add new tuple with that s_id and new advisor

1. Transactions

- *Transaction*: A sequence of database actions enclosed within special tags
- Properties:
  - ***Atomicity***: Entire transaction or nothing
  - ***Consistency***: Transaction, executed completely, takes database from one consistent state to another
  - ***Isolation***: Concurrent transactions *appear* to run in isolation
  - ***Durability***: Effects of committed transactions are not lost

- Consistency: Transaction programmer needs to guarantee that
  - DBMS can do a few things, e.g., enforce constraints on the data

- Rest: DBMS guarantees

2. MongoDB: A Document Database

MongoDB	DBMS
Database	Database
Collection	Relation
Document	Row/Record
Field	Column

Document = {…, field: value, …}

Where value can be:

- Atomic
- A document
- An array of atomic values
- An array of documents

{ qty : 1, status : "D", size : {h : 14, w : 21}, tags : ["a", "b"] },

Can also mix and match, e.g., array of atomics and documents, or array of arrays
[Same as the JSON data model]

Internally stored as BSON = Binary JSON

- Client libraries can directly operate on this natively

2. MongoDB Retrieval Queries

find() = SELECT <projection>
FROM Collection
WHERE <predicate>
limit() = LIMIT
sort() = ORDER BY

```
db.inventory.find(
        { tags : red },
        {_id : 0, instock : 0} )
.sort ( { "dim.0": -1, item: 1 } )
.limit (2)
```

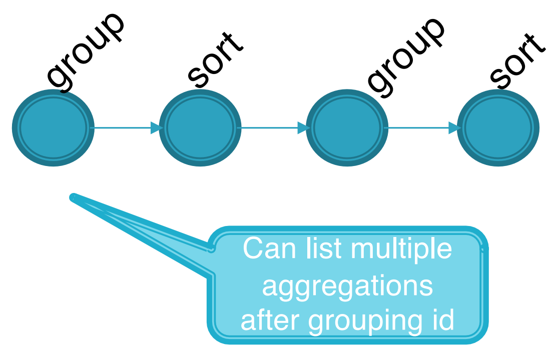
FROM
WHERE
SELECT
ORDER BY
LIMIT

2. A More Complex Query

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

Find, for every state, the biggest city and its population

aggregate([
{ $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
{ $sort: { pop: -1 } },
{ $group: { _id : "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } } },
{ $sort : {bigPop : -1} }
])



Approach:

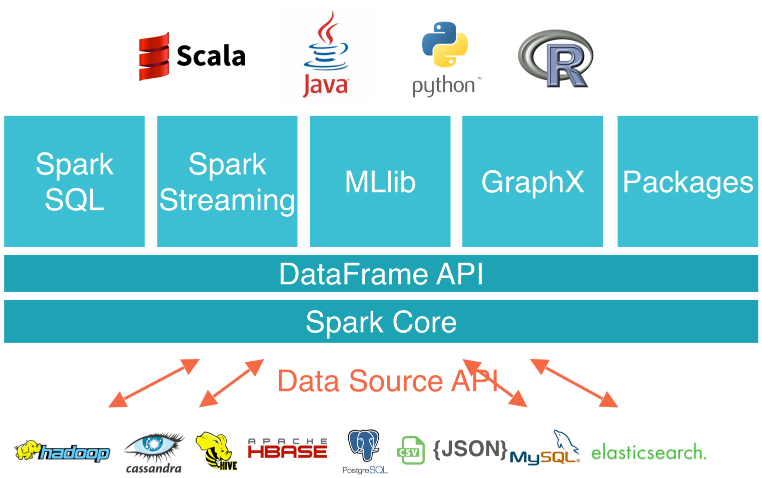
- Group by pair of city and state, and compute population per city
- Order by population descending
- Group by state, and find first city and population per group (i.e., the highest population city)
- Order by population descending

{ "_id" : "IL", "bigCity" : "CHICAGO", "bigPop" : 2452177 }
{ "_id" : "NY", "bigCity" : "BROOKLYN", "bigPop" : 2300504 }
{ "_id" : "CA", "bigCity" : "LOS ANGELES", "bigPop" : 2102295 }
{ "_id" : "TX", "bigCity" : "HOUSTON", "bigPop" : 2095918 }
{ "_id" : "PA", "bigCity" : "PHILADELPHIA", "bigPop" : 1610956 }
{ "_id" : "MI", "bigCity" : "DETROIT", "bigPop" : 963243 }
...

Syntax somewhat different when called from within Python3 (using pymongo)

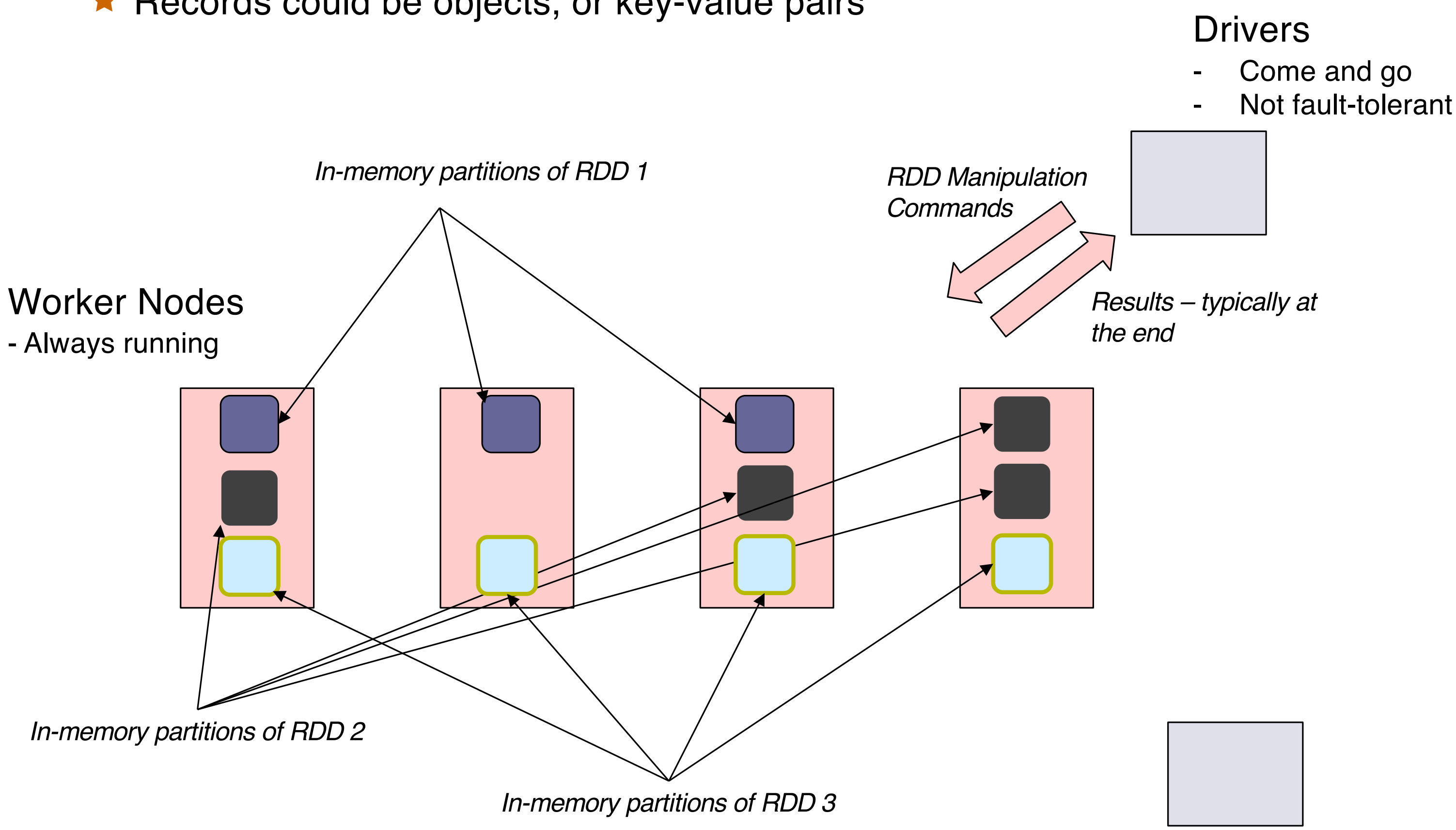
3. Spark

- Open-source, distributed cluster computing framework
- Much better performance than Hadoop MapReduce through in-memory caching and pipelining
- Originally provided a low-level RDD-centric API, but today, most of the use is through the “Dataframes” (i.e., relations) API
  - Dataframes support relational operations like Joins, Aggregates, etc.



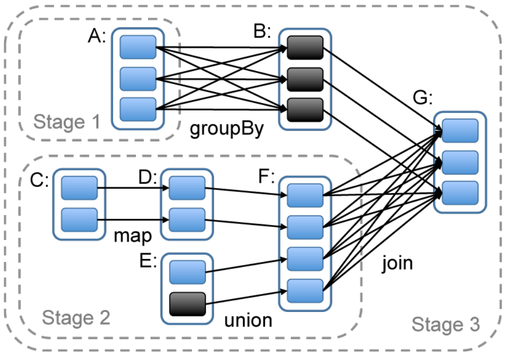
3. Resilient Distributed Dataset (RDD)

- **RDD =** Collection of records stored across multiple machines in-memory
  - Records could be objects, or key-value pairs



3. Spark

- Why "Resilient"?
  - Can survive the failure of a worker node
  - Spark maintains a "lineage graph" of how each RDD partition was created
  - If a worker node fails, the partitions are recreated from its inputs
  - Only a small set of well-defined operations are permitted on the RDDs
    - But the operations usually take in arbitrary "map" and "reduce" functions



- Fault tolerance for the "driver" is trickier
  - Drivers have arbitrary logic (cf., the programs you are writing)
  - In some cases (e.g., Spark Streaming), you can do fault tolerance
  - But in general, driver failure requires a restart

3. Example Spark Program

Initialize RDD by reading the textFile and partitioning
If textFile stored on HDFS, it is already partitioned – just read each partition as a separate RDD partition

Split each line into words, creating an RDD of words

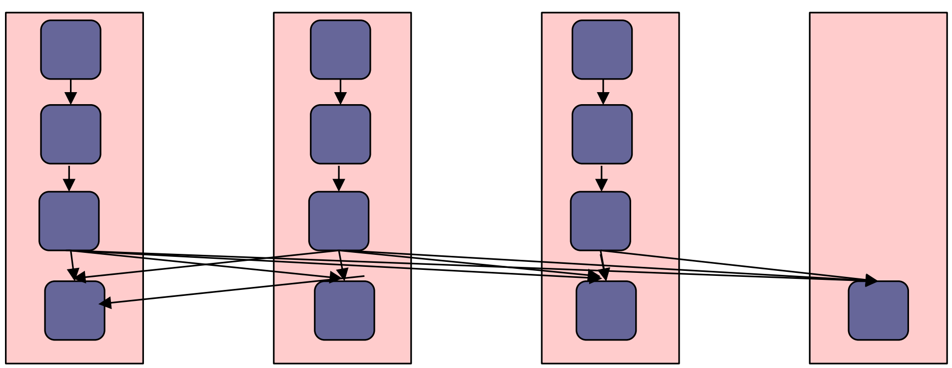
For each word, output (word, 1), creating a new RDD

Do a group-by SUM aggregate to count the number of times each word appears

Driver

```
from pyspark import SparkContext

sc = SparkContext("local", "Simple App")

textFile = sc.textFile("README.md")

counts = textFile
                 .flatMap(lambda line: line.split(" "))
                 .map(lambda word: (word, 1))
                 .reduceByKey(lambda a, b: a + b)

print(counts.take(100))
```

Retrieve 100 of the values in the final RDD



3. Spark

- Operations often take in a ”function” as input
- Using the inline “lambda” functionality

```
flatMap(lambda line: line.split(" "))
```

- Or a more explicit function declaration

```
def split(line):
      return line.split(" ")

flatMap(split)
```

- Similarly ”reduce” functions essentially tell Spark how to do pairwise aggregation

```
reduceByKey(lambda a, b: a + b)
```

  - Spark will apply this to the dataset pair of values at a time
  - Difficult to do something like “median”

Outline

- Examples of Data Management Systems
- **Overview**
- Architecture of Database System

Data Management Systems

- Massively successful for *highly structured or semi-structured data*
  - Why ? Structure in the data (if any) can be exploited for ease of use and efficiency

  - How ?

  - Two Key Concepts:
    - Data Modeling: Allows reasoning about the data at a high level
      - e.g. “emails” have “sender”, “receiver”, “…”
      - Once we can describe the data, we can start “querying” it
    - Data Abstraction/Independence:
      - Layer the system so that the users/applications are insulated from the low-level details

Data Modeling

- Data modeling
  - **Data model**: A collection of concepts that describes how data is represented and accessed
  - **Schema:** A description of a specific collection of data, using a given data model

  - Some examples of data models
    - Relational, Entity-relationship model, XML…
    - Object-oriented, object-relational, semantic data model, RDF…

  - Why so many models ?
    - Tension between descriptive power and ease of use/efficiency
    - More powerful models → more data can be represented
    - More powerful models → harder to use, to query, and less efficient

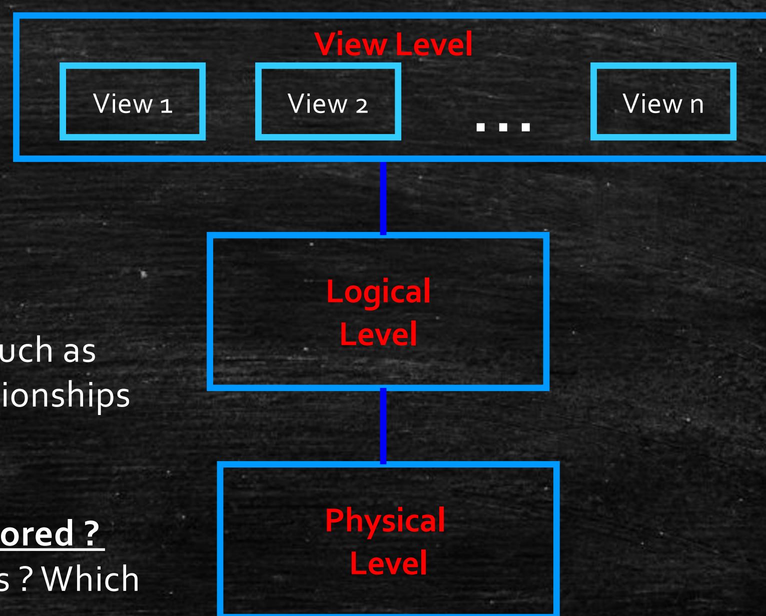
Data Abstraction/Independence

Hiding *low-level details* from the users of the system

What data users and application programs see ?

What data is stored ?
describe data properties such as data semantics, data relationships

How data is actually stored ?
e.g. are we using disks ? Which file system ?



Design Dimensions for a DMS

- User-facing
  - Data Model
  - Query Language and/or Programming Framework
  - Transactions
  - Performance Guarantees/Focus
  - Consistency Guarantees

These ”define” the “type” of the database

- Implementation
  - In-memory and at-rest storage representations
  - Target Computational Environment
  - Query processing and optimization
  - Transactions’ implementation
  - Support for streaming, versioning, approximations, etc.

Query Languages/Frameworks

- Define how to go from input data, to some desired output
  - Depends to some extent on the data model, but still a lot of flexibility
- Want this to be as ”high-level” or “declarative” as possible
  - Too high-level → fewer use cases will be covered
  - Too low-level → harder to use, support or optimize
  - Lot of work on trying to find the “right” level of abstraction
  - Interest in formally defining the power of a language, etc.
- Examples:
  - SQL: Input relations → output relations
  - Apache Spark RDD or Map-Reduce: Input “set of objects” → output ”set of objects”
  - BlinkDB: Input relations + approximation guarantees → output relations
  - Visualization Tools: Input datasets → Plots
- If supporting “streaming” or “versioning” or “approximations”, need to define what that means

Transactions/Updates (User-facing)

- Support for updating the data in the DMS
  - Some of the same issues as query language w.r.t. the expressiveness of the language
- Some considerations:
  - Consistency guarantees around updates (ACID or not)
    - Becomes more complicated in the distributed setting, with replication and sharding/partitioning
  - Batch updates vs one-at-a-time (impact on staleness)
  - Immutability: guarantees around no-tampering (e.g., blockchains)
  - Versioning: ability to support multiple branches, and ”time-travel”
- If the language is not expressive enough, have to do more work in the applications → impact on guarantees
  - e.g., MongoDB (and many other NoSQL stores) didn’t support multi-collection updates for a long time

In-memory and at-rest storage representations

- How is data laid out on disks (at rest) and in-memory, and across machines
  - Significant impact on performance
  - Depends somewhat on data model, but not fully ("Data Independence")
  - May use different representations when loading in memory (serialization/deserialization cost)
  - Usually we also build "indexes" for efficient search
  - Transmission over network also a concern
- Some options:
  - Row-oriented storage for relational model
    - Traditional approach: good for updates but bad for queries
  - Column-oriented storage for relational model
    - Really good performance for queries, but updates not easy to handle
  - Object storage (e.g., with pointers) for object-oriented databases or Graph databases
    - Pointers don't translate from disk to memory easily
  - Hierarchical storage for JSON/XML
  - Structured file formats like CSV (row), Parquet (columnar) for Data Lakes
    - Less up-front cost of "ingesting" the data, but more complex and less efficient to support
    - Harder to put any "structure" or "data model" on top of it
- Thoughts:
  - Cost of "ingest" must be amortized over many uses – for one-time use of data, prefer to leave in its native format

Target Computational Environment

- Many, many combinations here
  - Single machine vs parallel (locally) vs geographically distributed
  - Hardware
    - e.g., multi-core vs many-core, large-memory, disks or SSDs, RDMA, cache assumptions, and so on
  - Use of cloud/virtualization
    - Can have a significant impact on performance guarantees
    - Also, may put limits on what can be done (e.g., if using "serverless functions")
- Hard to build a different system for each combination
- Increasing interest in "auto-tuning" through use of ML
  - Try to "learn" how to do things for a new environment

Query Processing and Optimization

- Depends significantly on how “declarative” is the query language/framework
- Most systems support a collection of low-level “operators”
  - Relational: joins, aggregates, etc.
  - Apache Spark: map, reduce, joins, group-by, …
- Should choose a good set of operators
  - Restricts the optimization abilities
  - e.g., if only support “binary” joins then lose the ability to optimize multi-way joins
  - In general, a sequence of operations will perform worse than a single equivalent operation
- Need to map from the overall “task” or “query” into those low-level operators
  - Usually called a “query execution/evaluation plan”
  - There may potentially be many many ways to do this (depending on how declarative)
  - Try to choose in a “cost-based” manner
    - Need the ability to estimate costs of different plans
    - ”Heuristics” often preferred in less mature systems

Query Processing and Optimization

- Cost measure
  - Important to decide what resource you are optimizing
  - Need to focus on the bottlenecks of the environment
  - Traditionally: CPU, Memory, Disks
  - Today, network costs play a very important role
  - Also: optimizing for "total resources" or "wall-clock time" ?
    - Especially important in parallel/distributed environments
- May wish to "pre-compute" certain queries to reduce the query execution times
  - Especially for "real-time" queries over "streaming" data
  - Often called "materialized views" in the context of relational databases
  - Any pre-computed data must be kept up-to-date
- Adaptive query processing
  - May wish to "change" the query plan during execution based on what we are seeing

Support for Streaming, Versioning, Approximations, etc…

- Streaming
  - Usually need to keep a lot of pre-built state to handle high-rate data streams
  - Each new update → modify the pre-built state, and output results
  - Hard to do this in a generic way
    - A specialized system will likely have much lower response times (e.g., in financial settings)
- Versioning
  - So far, the focus has primarily been on storage (i.e., how to compactly store the version history over time)
  - The "retrieval" of old versions considered less important to date
- Immutability
  - More interest in recent years on this, but still pretty open from a database perspective
- Approximate Query Processing
  - Usually need additional constructs like "random samples"

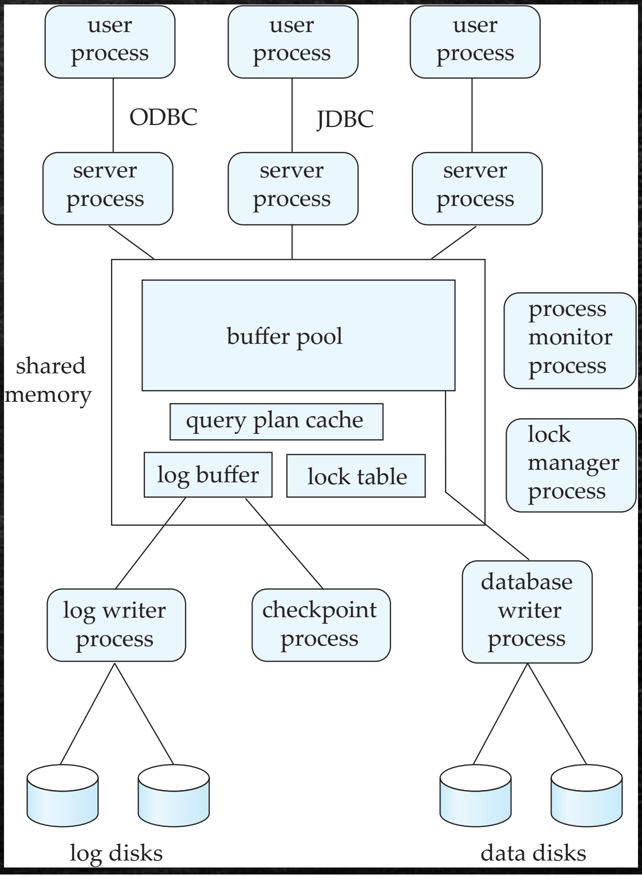
Recap

- Not intended to cover all data management research, but as a helpful guide to think about data management systems
  - Data cleaning, visualizations, security, privacy, …
- Finding the right abstractions is often the key to wide usage
- More complex abstractions may provide short-term wins, but often become difficult to manage and use over time
- Implementations have become very complex and involved today
  - Easy to obtain significant benefits focusing on a specific workload and hardware
  - But hard to get, and/or reason about performance in general settings
  - Experimental evaluations can't cover all different scenarios

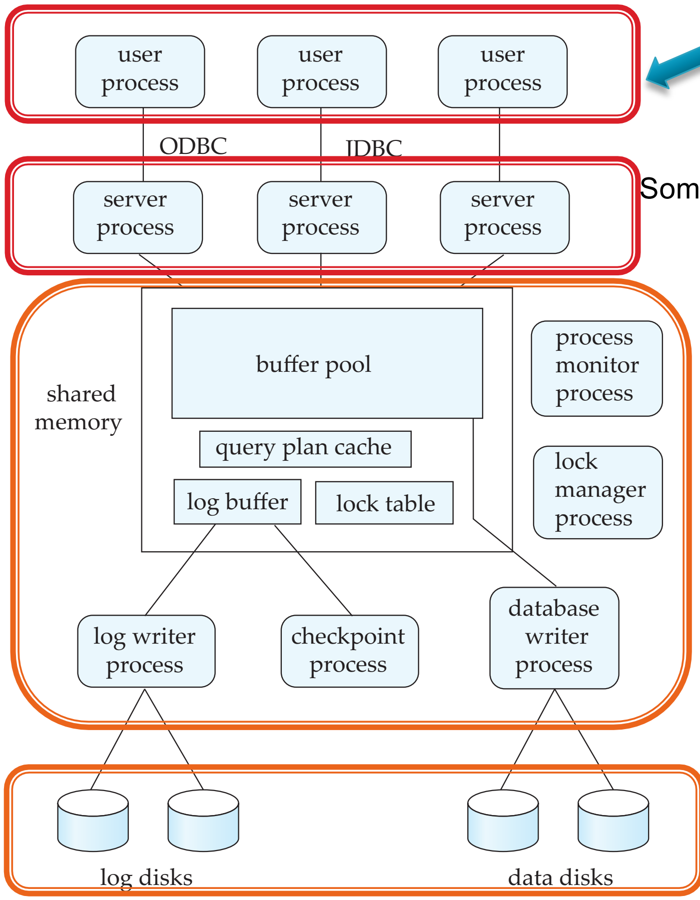
Database Architecture: Pre-2000's

- All data was typically in hard disks or arrays of hard disks
- RAM (Memory) was never enough
  - So always had to worry about what was in memory vs not
- Almost no real "distributed" execution
  - Different from "parallel", i.e., on co-located clusters of computers
- Relatively well-understood use cases
  - Report generation
  - Interactive data analysis and exploration
  - Supporting transactions

From Chapter 20
Database System Concepts (7th Edition)



Traditional RDBMS Architecture



Clients may be anywhere – e.g., ATMs, desktops, laptops, web apps etc.

Talk to the database using standard protocols like JDBC/ODBC, SOAP, or REST (today), or proprietary protocols

Some sort of load balancer or intake mechanism

Typical components in a database system: some for queries, some for transactions

Maybe on a single physical computer or a cluster connected by a fast network

Data Storage Systems:
(1) Punch cards (long time ago)
(2) Hard disks (still prevalent)
(3) SSDs

Need "redundancy" and "fault-tolerance"
Data once stored should always be there

RAID = Redundant Array of Independent Disks

Database Architecture: Today

- Much more diversity in the architectures that we see
  - More modern hardware architectures
    - Massively parallel computers
    - SSDs
    - Massive amounts of RAM – often don't need to worry about data fitting in memory
    - Much faster networks, even over a wide area
    - Virtualization and Containerization
    - Cloud Computing
  - As a result: Data and execution typically distributed all over the place
- Much more diversity in data processing applications
  - Much more non-relational data (images, text, video)
  - Data Analytics/Machine learning more common use-cases
- Much more diversity in "data models"
  - Document data models (JSON, XML), Key-value data model, Graph data model, RDF

Data Warehouses

For: Large-scale data processing (TBs to PBs)

Parallel architectures (lots of co-located computers)

SQL and Reporting

No transactions

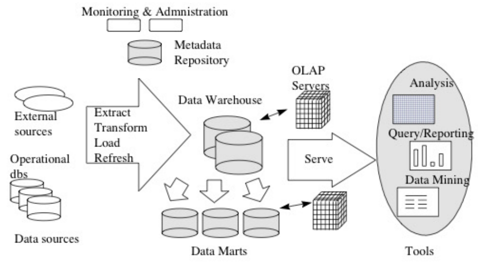


Figure 1. Data Warehousing Architecture

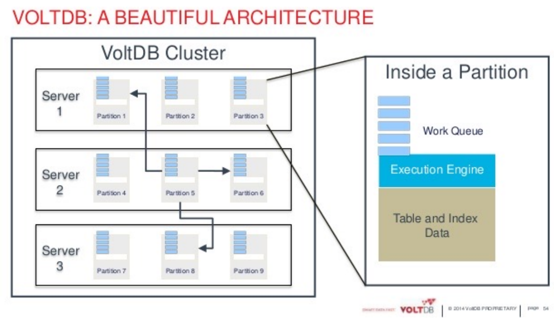
In-memory OLTP (on-line transaction processing)

For: Extremely fast transactions

Many-core or parallel architectures

Very limited SQL – mostly focused on "writes"

Typically assume data fits in memory across servers



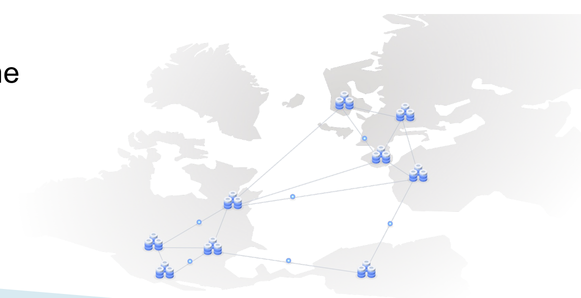
Highly available, distributed OLTP

For: Distributed scenarios where clients are all over the world

Focus on "consistency" – how to make sure all users see the same data

Limited SQL – mostly focused on "writes"

Considerations of memory vs disk less important

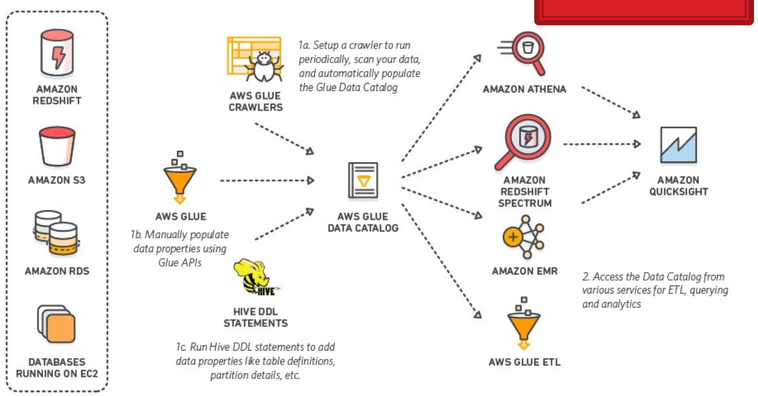


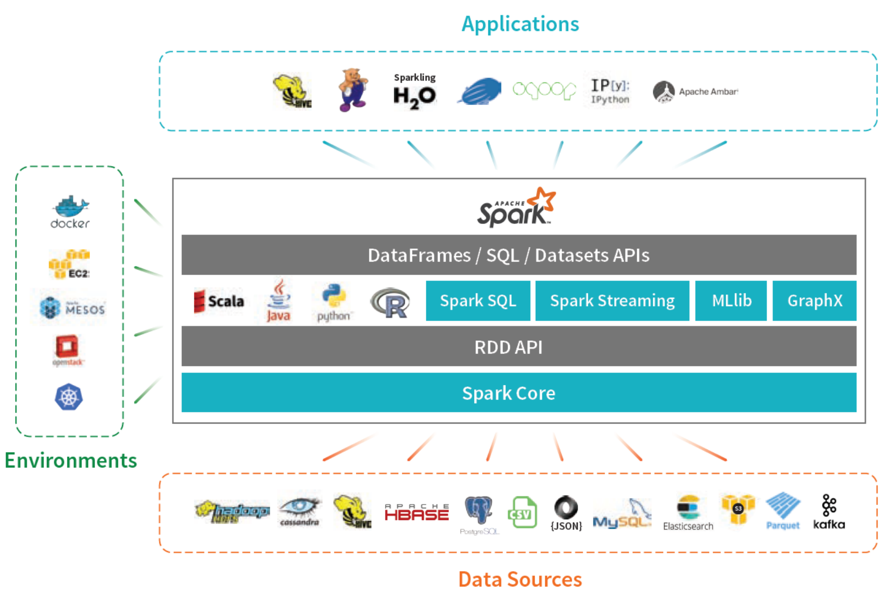
Extract-Transform-Load Systems, or Map-Reduce, or Big Data Frameworks

For: Large-scale, “ad hoc” data analysis

Mix of parallel and distributed architectures
Data usually coming from many different sources
Mix of SQL, Machine Learning, and ad hoc tasks (e.g., do image analysis, followed by SQL)

AWS Glue





Apache Spark

Recap

- Key takeaway: Modern data architectures are all over the place
- Fundamentals haven't changed that much though
  - We are still either:
    - Going from some "input datasets" to an "output dataset" (queries/analytics)
    - Modifying data (transactions)
  - SQL is still very common, albeit often disguised
    - Spark RDD operations map nicely to SQL joins and aggregates (unified now)
    - MongoDB lookups, filters, and aggregates map to joins, selects, and aggregates in SQL
- But "performance trade-offs" are all over the place now
  - 30 years ago, we worried a lot about hard disks and things fitting in memory
  - Today, focus more on networks
- Focus has shifted to other aspects of data processing pipelines
  - Analytics/Machine learning, data cleaning, statistics