No-SQL databases – Intro

Not in your textbook

Why go beyond textbook?

- What we teach in 484
  - Concepts, techniques, and ideas
- What we do NOT teach in 484
  - Specific tools and technologies
- Why I am teaching you this lecture?
  - With 484 concepts, you can easily learn state-of-the-art
  - Prepare you for the job market
  - Understanding where the DB industry is headed

Overview

- Traditional RDBMS
  - Most of the focus of this course

- Modern RDBMS (performance improvement strategies, based on workload) – may get to some of this in part 2 of the course
  - In-memory DBs
  - Columnar DBs
  - Approximate DBs
- NoSQL
  - Key-value stores
  - Document stores
  - MapReduce

NoSQL

- Key observation: Not every data problem is best solved by traditional relational databases
- NoSQL = No SQL = Not using traditional RDBMS
    - NoSQL ≠ Not using SQL language!
    - NoSQL = Not Only SQL
        - They may still support SQL language, e.g., Hive!

Traditional RDBMS vs. NoSQL

A DBMS provides: efficient, reliable, convenient, and safe multi-user storage of and access to massive amounts of persistent data.

- Convenient
  - Simple data model
  - Declarative query languauge
- Multi-user
  - Transaction guarantees (ACID)
- Safe
- Persistent
- Reliable
- Massive
- Efficient

NoSQL

- ~~Convenient~~
  - ~~Simple data model~~
  - ~~Declarative query languauge~~
- **Multi-user** ✓✓✓✓
  - ~~Transaction guarantees (ACID)~~
- Safe
- Persistent
- ~~Reliable:~~ redoing is OK
- **Massive** ✓✓✓
- **Efficient** ✓✓

DB-engines.com Ranking

Rank Sep 2024	Rank Aug 2024	Rank Sep 2023	DBMS	Database Model	Score Sep 2024	Score Aug 2024	Score Sep 2023
1.	1.	1.	Oracle	Relational, Multi-model	1286.59	+28.11	+45.72
2.	2.	2.	MySQL	Relational, Multi-model	1029.49	+2.63	-82.00
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	807.76	-7.41	-94.45
4.	4.	4.	PostgreSQL	Relational, Multi-model	644.36	+6.97	+23.61
5.	5.	5.	MongoDB	Document, Multi-model	410.24	-10.74	-29.18
6.	6.	6.	Redis	Key-value, Multi-model	149.43	-3.28	-14.26
7.	7.	↑ 11.	Snowflake	Relational	133.72	-2.25	+12.83
8.	8.	↓ 7.	Elasticsearch	Multi-model	128.79	-1.04	-10.20
9.	9.	↓ 8.	IBM Db2	Relational, Multi-model	123.05	+0.04	-13.67
10.	10.	↓ 9.	SQLite	Relational	103.35	-1.44	-25.85
11.	11.	↑ 12.	Apache Cassandra	Wide column, Multi-model	98.94	+1.94	-11.11
12.	12.	↓ 10.	Microsoft Access	Relational	93.76	-2.61	-34.81
13.	13.	↑ 14.	Splunk	Search engine	93.02	-3.08	+1.63
14.	↑ 15.	↑ 17.	Databricks	Multi-model	84.24	-0.22	+9.06
15.	↓ 14.	↓ 13.	MariaDB	Relational, Multi-model	83.44	-3.09	-17.01
16.	16.	↓ 15.	Microsoft Azure SQL Database	Relational, Multi-model	72.95	-2.08	-9.78
17.	17.	↓ 16.	Amazon DynamoDB	Multi-model	70.06	+1.15	-10.85
18.	↑ 19.	18.	Apache Hive	Relational	53.07	-2.17	-18.76
19.	↓ 18.	↑ 20.	Google BigQuery	Relational	52.67	-2.86	-3.80
20.	20.	↑ 21.	FileMaker	Relational	45.20	-1.47	-8.40

Overview

- Traditional RDBMS
- NoSQL
  1. Distributed Hash Tables
     - Key-value stores
     - Document stores
  2. MapReduce
     - Hadoop
     - Spark
     - SQL-on-MapReduce
  3. ~~Graph engines~~ We won't talk about these

Distributed Hash Table

- Given key, find value
- Hash table itself is big, and we don’t want to centralize the look up (can become a bottleneck).
- Partition the hash table, and distribute across multiple nodes.
- Query comes in to some node, and is redirected to the correct one (possibly in multiple hops).

Key-Value Stores

- Simplest type of Data Model: (key, value) pairs
  - Examples: Redis, Amazon DynamoDB, Azure Cosmos DB, RocksDB
  - Value can be a binary string, ...
  - API: get(key), put(key, value), delete(key)
- Wide Column Stores: Support tables by storing
  - (key, columnName, value) triplets
  - Unlike a relational DB, the names and format of the columns can vary from one row to another within the same table
  - Examples: Cassandra, BigTable, HBase, ...

Wide Column Stores

- Use one (key, columnname, value) triple per logical column in a logical record.
- Not all columns are required in all records. Schema-free
- Typically, very large number of columns (possibly millions)
- Examples: Cassandra, BigTable, HBase, ... Figure below from the Google's BigTable paper

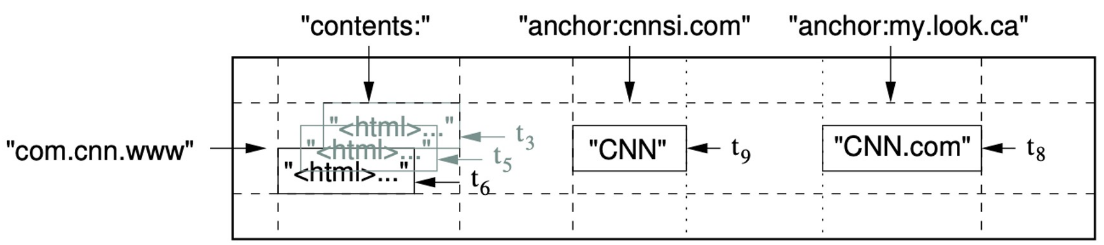


Figure 1: A slice of an example table that stores Web pages. The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN's home page is referenced by both the Sports Illustrated and the MY-look home pages, so the row contains columns named anchor:cnnsi.com and anchor:my.look.ca. Each anchor cell has one version; the contents column has three versions, at timestamps t_3, t_5, and t_6.

Document Stores

- Data model: (key, document) pairs
- Document: JSON format typically (or XML in some)
- In addition to look up by key, they can also fetch documents by their content
- Examples: MongoDB, CouchDB, DynamoDB (as well), DataBricks
- Can use multiple servers via sharding
- The DBMS provides a distributed hash table (DHT) – whicht allows look up of a document given a key.

Case Study: MongoDB Documents

- Document corresponds to a tuple in a relation

```
{
  name: "sue",                      ← field: value
  age: 26,                          ← field: value
  status: "A",                      ← field: value
  groups: [ "news", "sports" ]      ← field: value
}
```

- Documents are simply JSON objects in JavaScript syntax, consisting of field:value pairs.
- Documents can be hierarchical – a value can be a JSON object or a list.

Case Study: MongoDB Collections

- A collection corresponds to a table in relational databases. It is a set of documents (common structure among documents in the same collection is **NOT** enforced!)

Key Commands in MongoDB

- db.collectionname.insert(json_object)
- db.collectionname.find(predicate)
- The best way to learn is to simply follow the tutorial. This will open an interactive shell over the web to try a few commands out:
  - http://docs.mongodb.org/manual/tutorial/getting-started/
- You can also install mongodb server locally on your computer. Then run “mongosh” to open an interactive shell.

Example

- Download and setup mongo on your computer. Import sample.json:
- % mongoimport --collection users --file sample.json --jsonArray

More generally, to import into a mongo database at a server with userid and password:

% mongoimport <dbname> --host <hostname> -u <userid> -p <password> --collection <collectionname> -- file <filename> --jsonArray

Example queries

- Type “mongo” to connect to local mongo.
- “SELECT * from users”: Mongo equivalent:

```
> db.users.find();
```

  - It returns a cursor object and prints 10 tuples at a time.
  - Type “it” to see additional tuples.
- Using Javascript variables:

```
> var x = db.users.find();
```

Iterate over a cursor

```
var mycursor  = db.users.find();
while (mycursor.hasNext()) {
	var w = mycursor.next(); // next document
	print(w.user_id, w.DOB); // print fields
}
// mycursor now points to the end.
```

Selections: Predicates in Find

Find users born on 21st Nov.

- var mycursor = db.users.find({"DOB" : 21, "MOB" : 11});

Find users born on 21st Nov. in state ”Rohan”:

> var mycursor = db.users.find({"DOB" : 21, "MOB" : 11, "hometown.state" : "Rohan"});

Note that the structure of the document is:

{user_id: …,

DOB: … ,

MOB : ... ,

hometown : {city : ... state : ..., country : ...},

...

}

Projections

- Find can also include projections.
- Find first_name and last_name of users born in state Rohan on Nov. 21st:

  var mycursor = db.users.find({"DOB" : 21, "MOB" : 11, "hometown.state" : "Rohan"}, **{first_name : 1, last_name : 1});**

- > mycursor
- { "_id" : ObjectId("5664e69d270b10887550707d"), "first_name" : "Isabel", "last_name" : "THOMAS" }
- { "_id" : ObjectId("5664e69d270b1088755071d0"), "first_name" : "Gimli", "last_name" : "ANDERSON" }
- { "_id" : ObjectId("5664f005270b108875507398"), "first_name" : "Isabel", "last_name" : "THOMAS" }
- { "_id" : ObjectId("5664f005270b1088755074ea"), "first_name" : "Gimli", "last_name" : "ANDERSON" }
- >

Projections

- _id is a special value, which serves as a key.
- Mongo automatically creates an _id value for inserted values in a collection. Dropping it in a projection requires an explicit projection to 0 for _id:

> var mycursor = db.users.find({"DOB" : 21, "MOB" : 11, "hometown.state" : "Rohan"}, {first_name : 1, last_name : 1, **_id : 0**});

> mycursor

{ "first_name" : "Isabel", "last_name" : "THOMAS" }

{ "first_name" : "Gimli", "last_name" : "ANDERSON" }

{ "first_name" : "Isabel", "last_name" : "THOMAS" }

{ ”first_name" : "Gimli", "last_name" : "ANDERSON" }

Counting

- Counting tuples: apply count() to results from find()

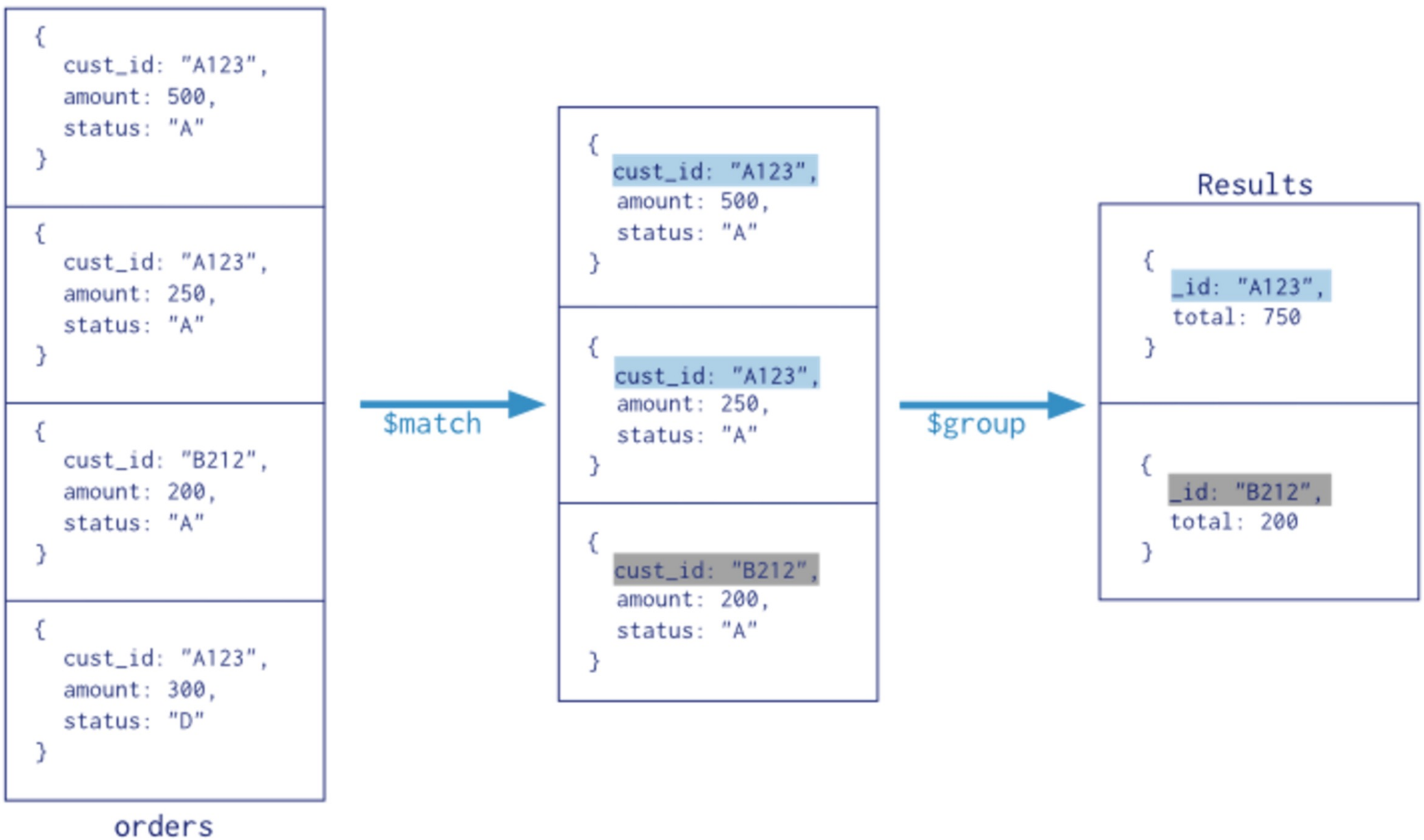
```
> var mycursor = db.users.find({"DOB" : 21, "MOB" : 11, "hometown.state" : "Rohan"}, {first_name : 1, last_name : 1});
> mycursor.count()
4
```

- The find command returned 4 documents.

Aggregations -- Pipeline

Collection

```
db.orders.aggregate( [
    $match stage ──►  { $match: { status: "A" } },
    $group stage ──►  { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
] )
```



Other Aggregate stages

- $group: similar to GROUP BY
- $sort : for sorting data
- $unwind <arrayfield>: flattens arrays. See docs.
- $out <out_collection>: to put the result into a output collection.

See documentation for other aggregate stages

Aggregate also returns a cursor.

MapReduce

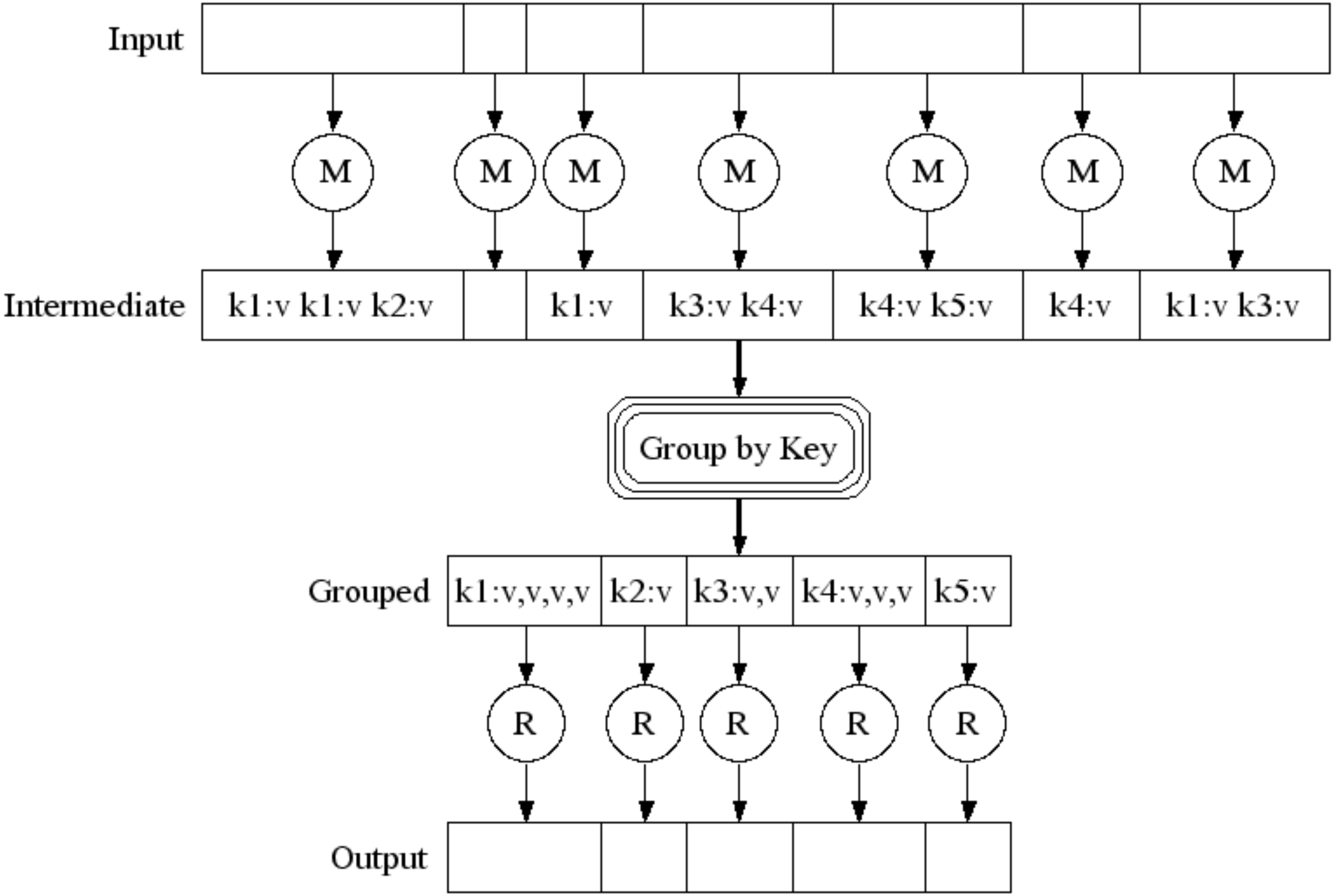
- MapReduce system provides:
  - Automatic parallelization & distribution
  - Fault-tolerance
  - Status & monitoring tools
  - Clean abstraction for programmers

Data-Centric Programming

- MapReduce has become very popular, for lots of good reasons
  - Easy to write distributed programs
  - Built-in reliability on large clusters
  - Bytestreams, not relations
  - “Schema-later”, or “schema-never”
  - Your choice of programming languages
  - Hadoop relatively easy to administer

MapReduce

- Many data programs can be written as *map* and *reduce* functions
- *map* transforms ***key, value*** inputs into new **`key'`**, **`value'`**
  - **`Map(k, v) => (k', v') list`**
- *reduce* receives all the vals for a given key' and can output to disk file
  - **`Reduce(k', v' list) => (out-key, out-val) list`**


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

Execution Pipeline

- MapReduce execution consists of 2 main stages:
  - Map
  - Reduce
- In stage 1, partition input data and run **map()** on many machines
- Then group intermediate data by key
- In stage 2, partition data by key and run **reduce()** on many machines
- Output is whatever reduce() emits

Processing Large Data

- Many CPUs needed. 1000s, not dozens
  - Programmer cannot know how many machines at program-time
  - Job is very long-lasting
  - Machines die, machines depart; job must survive
- Map/Reduce architecture makes it possible to handle all the above without burdening the programmer