**COMPSCI 590S Systems for Data Science** **Fall 2017**

# Lecture 5: DBMS and MapReduce

*Lecturer: Emery Berger* *Scribe(s): Aishwarya Turuvekere, Suraj Subraveti*

## 5.1 Review of MapReduce and Database performance

We reviewed papers that dealt with MapReduce, a fault tolerant programming paradigm for processing and creating huge datasets, built by Google. There was also another paper that described how the growing popularity of MapReduce based systems didn't necessarily mean that databases have become irrelevant. Both papers discussed their pros and cons, and suggested ways in which a hybrid system could leverage the advantages of both paradigms.

## 5.2 Pre-database systems

Before databases became a popular way for querying for data, it was stored as files on disk, and these had "records" of fixed length. To access anything in a file, one had to iterate through it using a cursor. That would involve writing code like this:

```
cursor c = file.begin();
while(cursor!=file.end()){
    String s = c.getNextRecord();
    //do something with the record
    c.moveToNextRecord();
}
```

For more complex operations, sorting the file on a particular field could be useful. For files that do not fit into the memory, some external sorting algorithm is required. This means that you would load chunks of the file into memory, and sort them, and merge these chunks until the file would be completely sorted, which is maybe okay if the data in the file is read only, but bad if there need to be updates performed, because one would need to sort the file each time something is updated. This costs a lot of I/O operations.

## 5.3 Databases

There are row and column store databases. Databases operate in an ACID fashion (Atomicity, Consistency, Isolation, Durability) via transactions. Prior to SQL (Structured Query Language) was QUEL, which became SEQUEL, then SQL. SQL is a declarative language. This means that the programmer states what they want done, and behind the scenes the interpreter determines the step(s) to complete the task(s). An example database with an imperative program instead of a declaritive query.

| Income | Name | Age |
|---|---|---|
| 23000 | Brown | 29 |
| | | .. |
| | | .. |

```
it = db.begin();
count = 0;
for( , it!db.end(); it++){
      if(it.getField("income")>100000){
         count++;
      }
   }
```

A query in a declaritive language will have a much different set of steps. For a query such as an insert, one might use a binary search to locate the correct location to insert a new record based on a given index or set of indices. This index could be represented as a B-tree. The result of these differences is that for tasks commonly done with a declaritive language like SQL, the imperative version of that program is much harder to write in a performative manner. Imperative languages can't hide details like index creation. C# and link use SQL-like code in an imperative language to avoid the problems of imperative code.
There is a standard for SQL, but each vendor has its own dialect of SQL. A SQL query will be of the general form:

```
SELECT...
FROM...
WHERE...
```

An example query would be:

```
SELECT (*)
FROM employee
WHERE employee.income > 100000;
```

When using SQL, one must be very explicit about the contents of the database. For this, the user specifies the content type of each column of each table in each database in what is known as a schema. The schema must be made prior to the creation of the table.
DBMS's don't have the advantage when it comes to semi-structured data. XML is a disaster. It encompasses HTML, but HTML on the web is rarely HTML compliant. Browsers tend to interpret HTML. MR can be a better fit in these cases because it is difficult to put webpages into databases.

### 5.3.1 Row based vs Column based

A database may be row based or column based. This is an implementation detail about the orientation of the data as it is stored on disk. A row based database will store an entire row contiguously on disk. This leads to a performance advantage when pulling an entire record or set of records from disk, and is straightforward to implement. This is also an advantage when distributing the database across multiple machines.
A column based database stores an entire column contiguously on disk. When accessing or appending entire rows, this will be slower than a row oriented database. However, there are significantly smaller read costs for certain access patterns (such as find rows where there is some condition). The database can avoid reading significant amounts of data compared to row based databases.
Whether the database is row or column based, it will have a query optimizer (except for SQLite) which will interpret the query, and then optimize the directed steps into the smallest number of steps which it calls a plan, and then executes the plan.

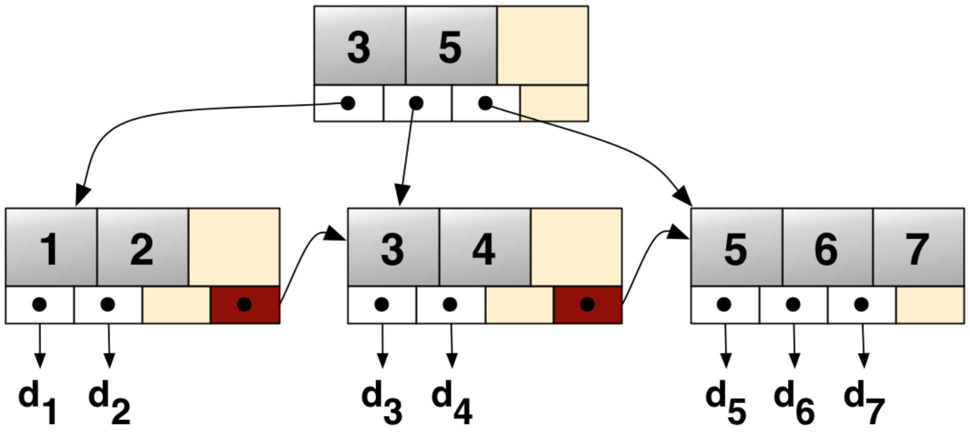

Figure 5.1: A B$^+$ tree

### 5.3.2 Indexes

Indices are used to speed up data access in tables. Updates need to propagate to indices, so indices increase the time it takes to make updates to a database. Problems with DBMSs come from scaling and updating indices. However, updates are hard with key-value stores. Deadlock can easily occur.

Using balanced binary trees like red black trees for indexing is good as it still provides a logarithmic look up time. But using such trees come with two problems:

- Space consumption: All the data is on a disk, and we want to access the disk as rarely as possible.
- Bad locality: As there is almost no locality of data with balanced binary trees, disk access tends to be more frequent.

B trees are a good alternative to balanced binary trees. Apart from the self balancing factor, we have huge gains due to the fact that it stores multiple keys per node. This reduces the number of disk accesses from $O(\log_2 n)$ to $O(\log_{nkeys} n)$. A related data structure is the B$^+$ trees in which all internal nodes have only keys. All values are stored in the leaves and are linked together like a linked list as shown in Figure 5.1

Although this seems like a great idea, and the intuitive thing to do would be to index all fields for faster lookups, it is usually not a good idea to do this, because indexes do take up significant space on the disk, and updates on these fields become slower. It is a bad tradeoff to index something that is more likely to be updated than read.Typically, database administrators decide what field(s) to index based on the query workloads and analysis of query logs.

### 5.3.3 Why disk access is expensive

As discussed above, disk access is expensive, and using data structures like B trees and $B^+$ trees that reduce the number of disk reads is a good idea. Conceptually, disks are relatively simple. Each disk platter has a flat circular shape, like a CD. The reads are usually spread all over which leads to latency in reads. There are two kinds of latency associated with disks:

- Seek latency: The time needed for moving the disk arm to the desired cylinder.
- Rotational latency: The time needed for the desired sector to rotate to the disk head.

Typical disks can transfer several megabytes of data per second, and they have seek times and rotational latency of several milliseconds.