Not Your Father's Database:

How to Use Apache Spark Properly in Your Big Data Architecture

Spark Summit East 2016

Not ~~Your Father's~~ Database:

How to Use Apache Spark Properly in Your Big Data Architecture

Spark Summit East 2016

databricks

About Me

2005 **Mobile Web & Voice Search**

About Me

2005 **Mobile Web & Voice Search**

2012 **Reporting & Analytics**

About Me

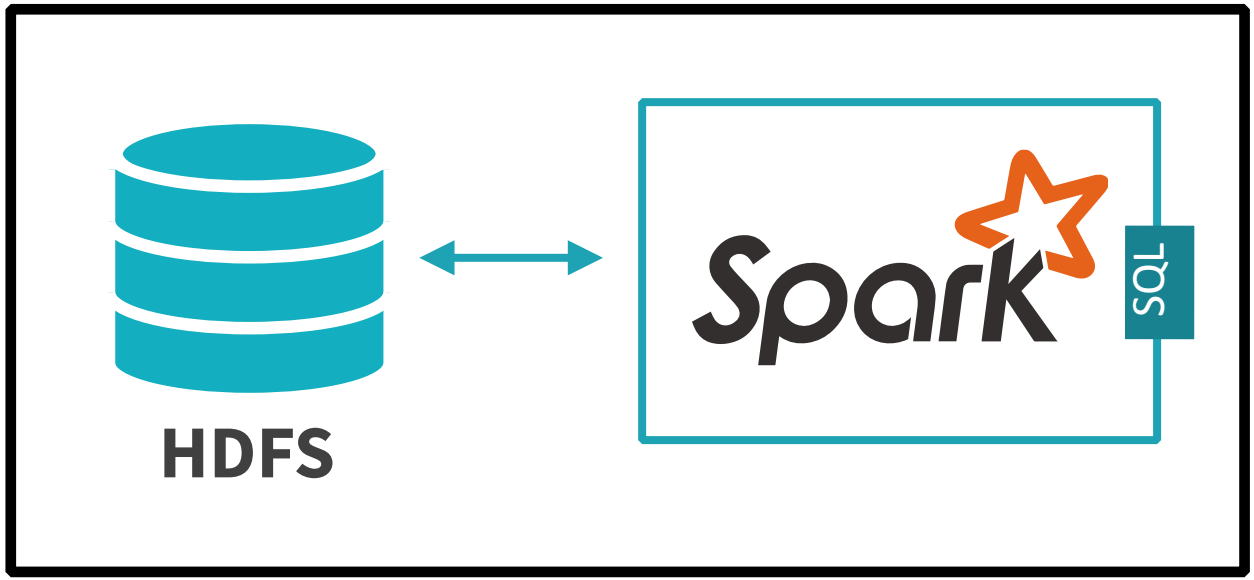
2005 **Mobile Web & Voice Search**

2012 **Reporting & Analytics**

2014 **Solutions Engineering**

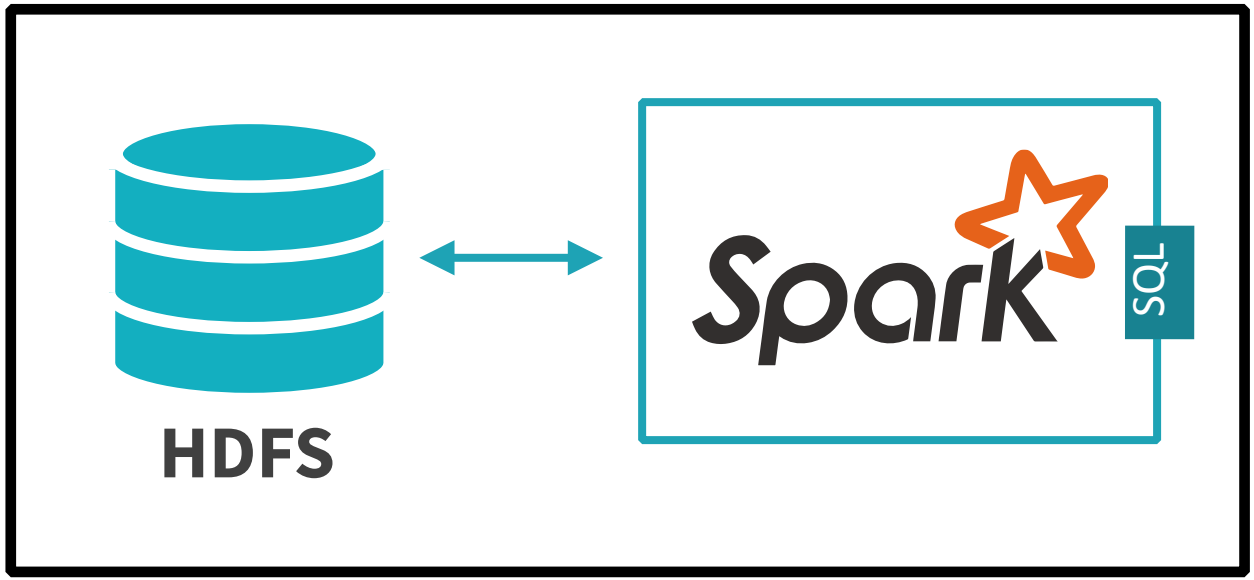
Is this your Spark infrastructure?

This system talks like a SQL Database…

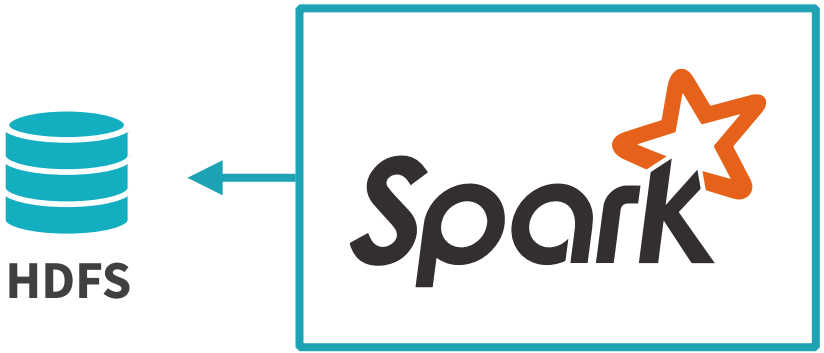


Is this your Spark infrastructure?

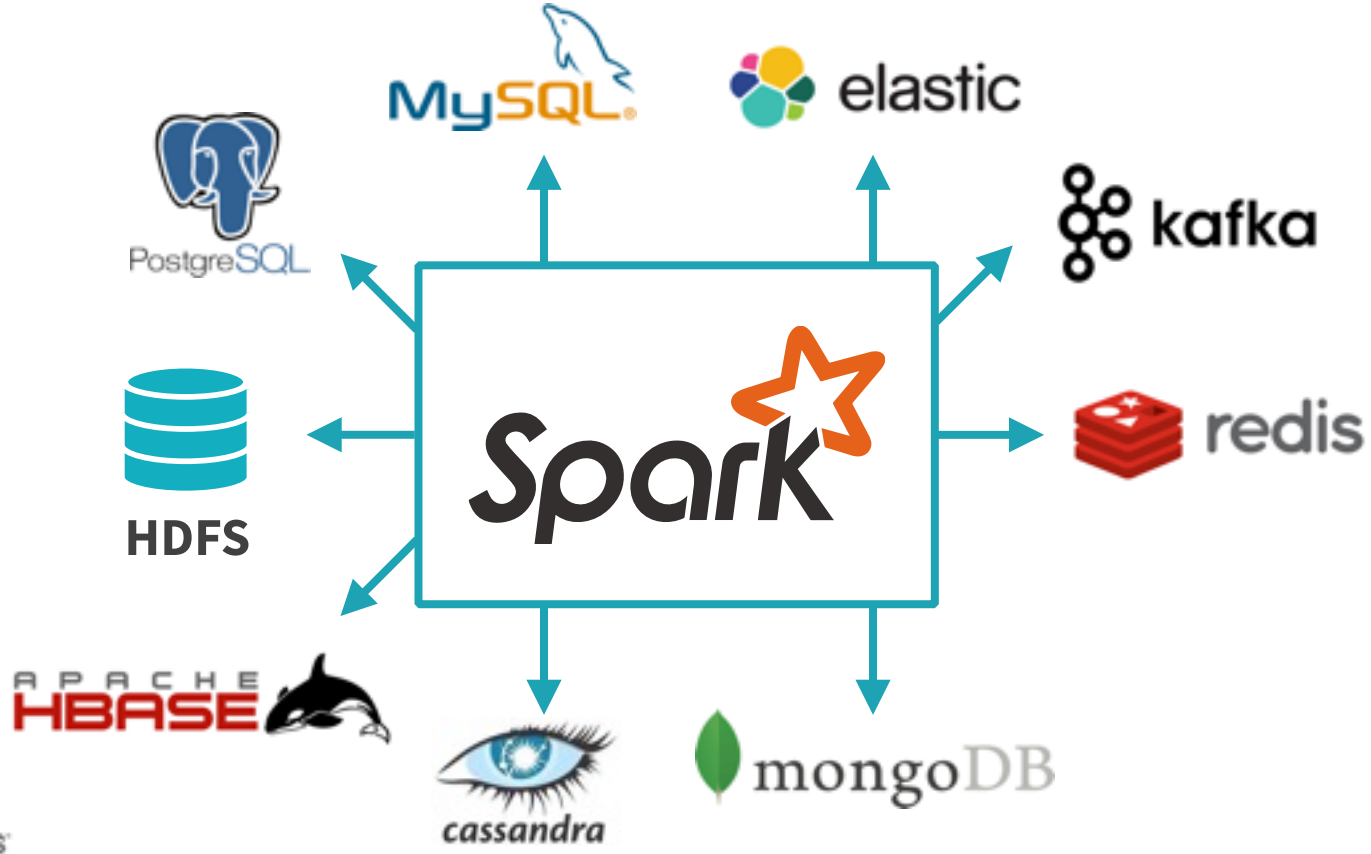
But the performance is very different…



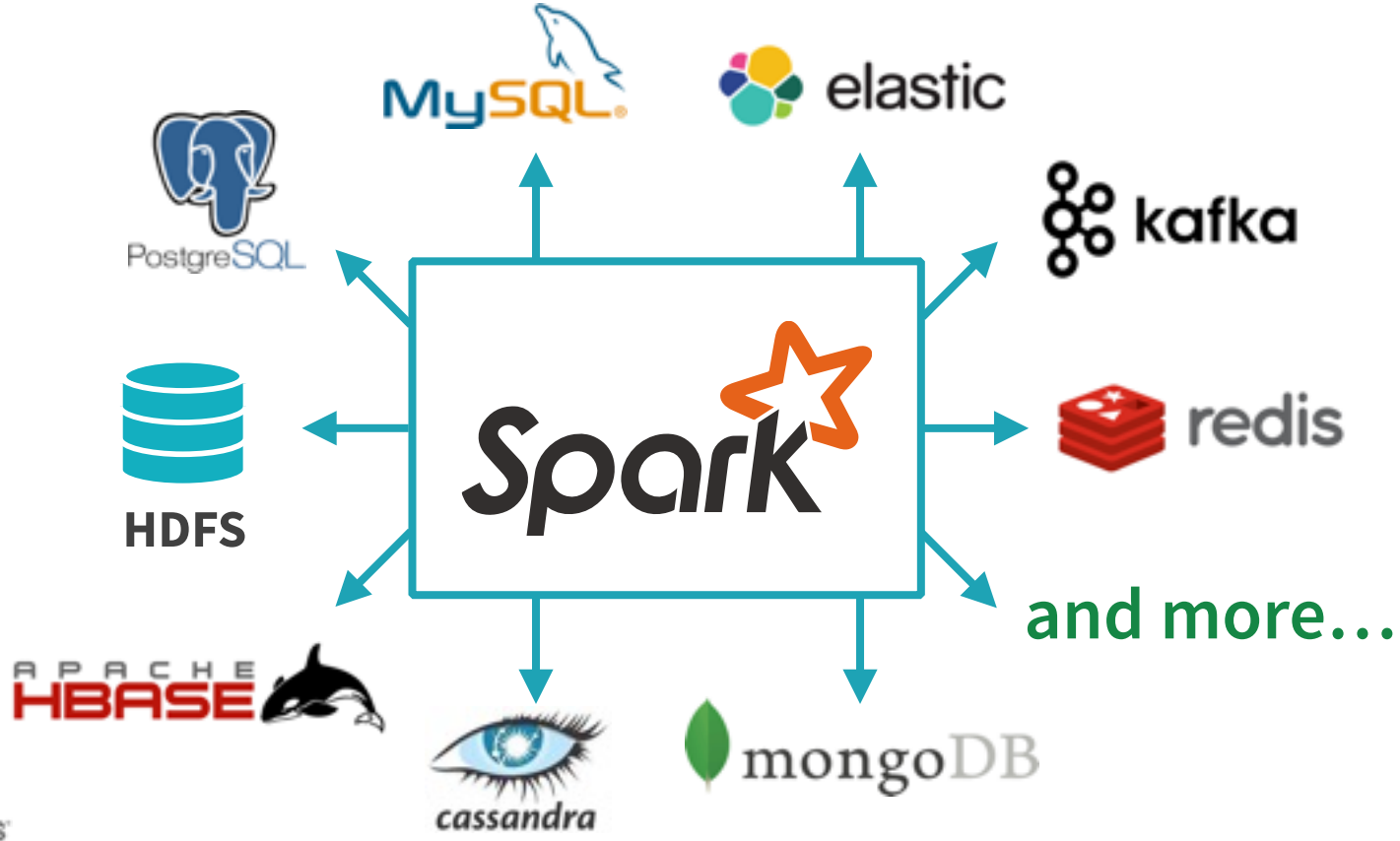
Just in Time Data Warehouse w/ Spark



Just in Time Data Warehouse w/ Spark



Just in Time Data Warehouse w/ Spark



Today's Goal

Know when to use other data stores besides file systems

Good: General Purpose Processing

Types of Data Sets to Store in File Systems:

- Archival Data
- Unstructured Data
- Social Media and other web datasets
- Backup copies of data stores

Good: General Purpose Processing

Types of workloads

- Batch Workloads
- Ad Hoc Analysis
  - Best Practice: Use in memory caching
- Multi-step Pipelines
- Iterative Workloads

Good: General Purpose Processing

Benefits:

- Inexpensive Storage
- Incredibly flexible processing
- **Speed and Scale**

Bad: Random Access

```
sqlContext.sql(
    “select * from my_large_table where id=2I34823”)
```

Will this command run in Spark?

Bad: Random Access

```
sqlContext.sql(
    “select * from my_large_table where id=2I34823”)
```

Will this command run in Spark?

Yes, but it’s not very efficient — Spark may have to go through all your files to find your row.

Bad: Random Access

Solution: If you frequently randomly access your data, use a database.

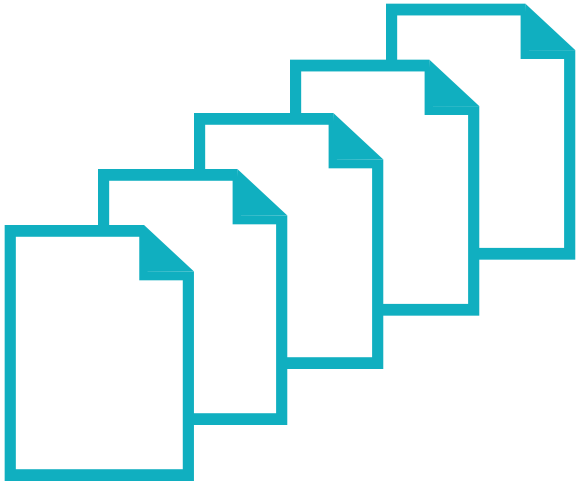
- For traditional SQL databases, create an index on your key column.
- Key-Value NOSQL stores retrieves the value of a key efficiently out of the box.

Bad: Frequent Inserts

sqlContext.sql(“insert into TABLE myTable
select fields from my2ndTable”)

Each insert creates a new file:

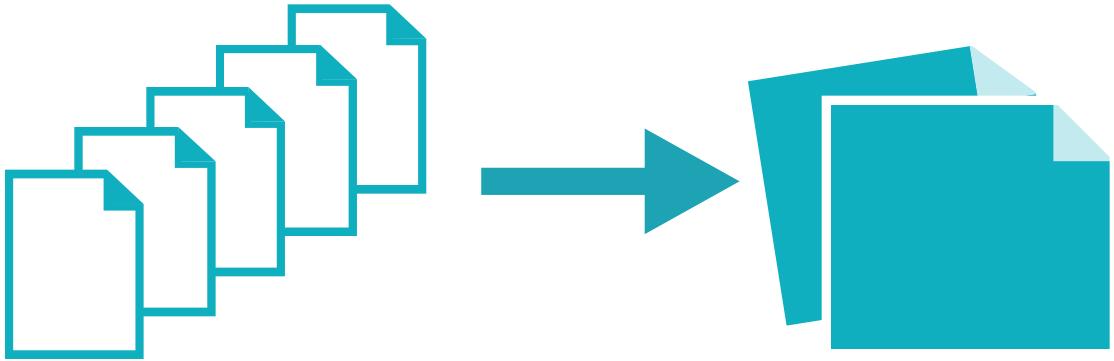
- Inserts are reasonably fast.
- But querying will be slow…



Bad: Frequent Inserts

Solution:

- Option 1: Use a database to support the inserts.
- Option 2: Routinely compact your Spark SQL table files.



Good: Data Transformation/ETL

Use Spark to splice and dice your data files any way:

File storage is cheap:

Not an “Anti-pattern” to duplicately store your data.

Bad: Frequent/Incremental Updates

Update statements — not supported yet.

Why not?

- **Random Access:** Locate the row(s) in the files.
- **Delete & Insert:** Delete the old row and insert a new one.
- **Update:** File formats aren't optimized for updating rows.

Solution: Many databases support efficient update operations.

Bad: Frequent/Incremental Updates

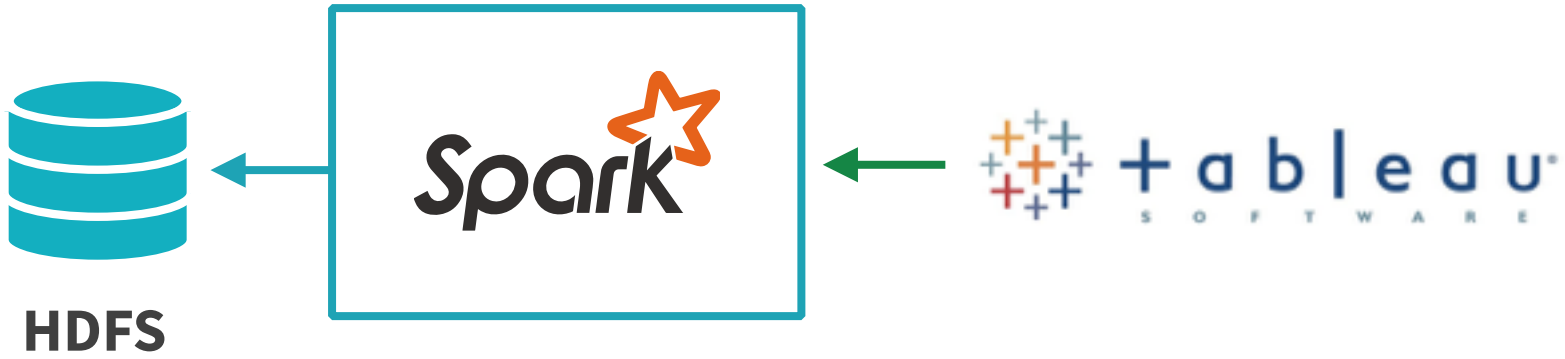
Use Case: Up-to-date, live views of your SQL tables.

Database Snapshot

Incremental SQL Query

Tip: Use ClusterBy for fast joins or Bucketing with 2.0.

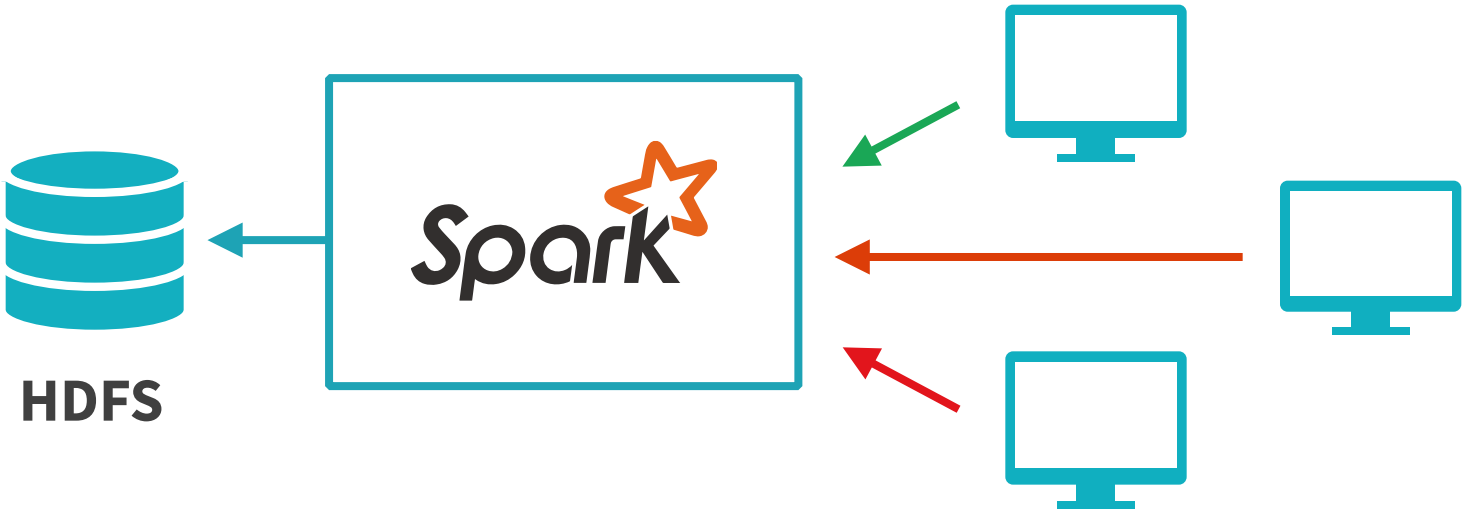
Good: Connecting BI Tools



Tip: Cache your tables for optimal performance.

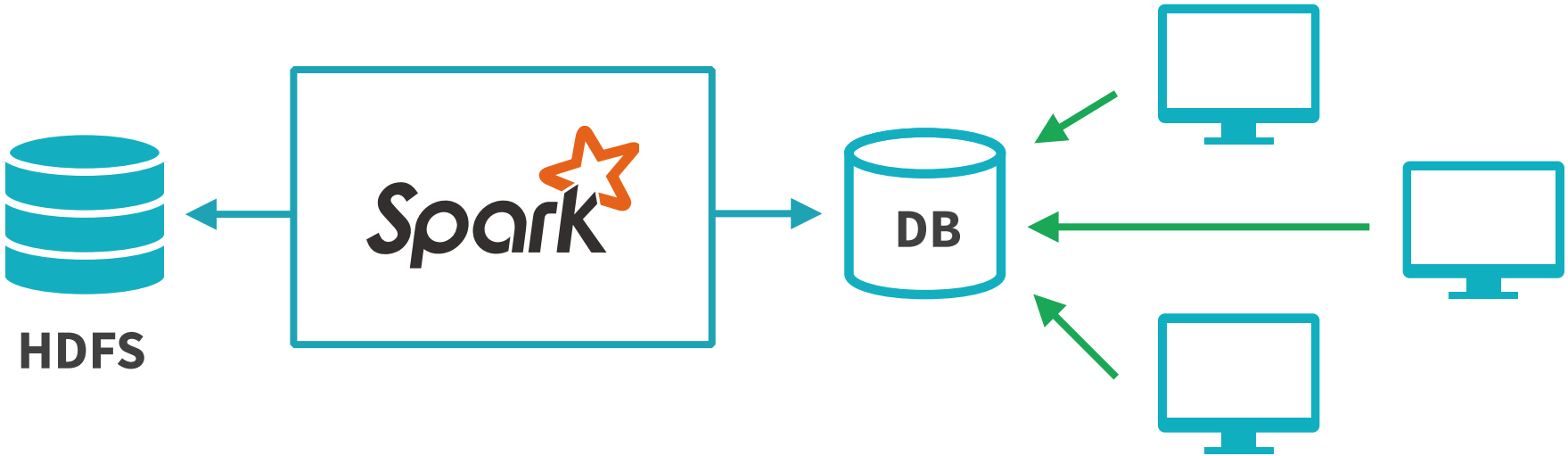
Bad: External Reporting w/ load

Too many concurrent requests will overload Spark.



Bad: External Reporting w/ load

Solution: Write out to a DB to handle load.



Good: Machine Learning & Data Science

Use MLlib, GraphX and Spark packages for machine learning and data science.

Benefits:

- Built in distributed algorithms.
- In memory capabilities for iterative workloads.
- Data cleansing, featurization, training, testing, etc.

Bad: Searching Content w/ load

```
sqlContext.sql(“select * from mytable
    where name like '%xyz%'”)
```

Spark will go through each row to find results.

Thank you