ASTER BACKGROUND
Our Founders

3 PhD students from Stanford C.S.

- Cool ideas...
- ... but no funding, no product, no clients!

OK, they had $10,000...
Our Product: \textit{nCluster}

- A massively scalable database designed for analytics.
- Runs on a cluster of commodity nodes.
- Scales from GBs to 100s of TBs and beyond.
- Standard SQL interface (via a command line tool, JDBC, ODBC, etc).
- Support MR-like functionality via user-defined SQL/MR functions.
Our Approach: Commodity Nodes

Query

Queen

Results

Server nodes
Processing + Storage
What are SQL/MR functions?

**SQL/MR functions:**
- Are Java functions meeting a particular API.
- Are compiled outside the database, installed via a command line tool, and then invoked via SQL.
- Take a database table of one schema as input and output rows back into the database.
- Are polymorphic. During initialization, a function is told the schema of its input (for example, (key, value)) and needs to return its output schema.
- Accept zero or more argument clauses (parameters), which can modify their behavior.
- Are designed to run on a massively parallel system by allowing the user to specify which slice of the data a particular instance of the function sees.
First Example: Word Count

**Problem:** Count the word frequency distribution across a set of documents.

**Input:** A database table containing the documents in question.

**Map Phase:** For each word in each document, outputs a row of the form (word, 1).

**Shuffle Phase:** Brings all rows with the same value for word together.

**Reduce Phase:** Count the number of rows for each word about output (word, <total-count>).
BEGIN;

CREATE FACT TABLE documents (body varchar,
    PARTITION KEY(body));

INSERT INTO documents VALUES ('this is a single test document. it is simple to count the words in this single document by hand. do we need a cluster?');

END;

SELECT body FROM documents;
public class tokenize implements RowFunction {
    ...

    public void operateOnSomeRows(RowIterator inputIterator,
                                   RowEmitter outputEmitter)
    {
        while ( inputIterator.advanceToNextRow() ) {
            String[] parts =
                splitPattern_.split( inputIterator.getStringAt(0) );

            for (String part : parts) {
                outputEmitter.addString(part);
                outputEmitter.addInt(1);
                outputEmitter.emitRow();
            }
        }
    }
}
public class count_tokens implements PartitionFunction {
    ...

    public void operateOnPartition(
        PartitionDefinition partitionDefinition,
        RowIterator inputIterator, RowEmitter outputEmitter)
    {
        int count = 0;
        String word = inputIterator.getStringAt(0);

        while ( inputIterator.advanceToNextRow() )
            count++;

        outputEmitter.addString(word);
        outputEmitter.addInt(count);
        outputEmitter.emitRow();
    }
}
BEGIN;

\install tokenize.jar
\install count_tokens.jar

SELECT word, count FROM count_tokens ( 
    ON ( SELECT word, count 
        FROM tokenize(ON documents)) 
    PARTITION BY word 
) ORDER BY word DESC;

ABORT;
Even Better: Forget the Reduce

BEGIN;

\install tokenize.jar

SELECT word, sum(count)
FROM tokenize(ON documents)
GROUP BY word
ORDER BY word;

ABORT;
Types of SQL/MR Functions

**RowFunction**
- Corresponds to a *map* function.
- Must implement the `operateOnSomeRows` method.
- Must be invoked without a PARTITION BY.
- “Sees” all the appropriate rows on a particular worker.

**PartitionFunction**
- Corresponds to a *reduce* function.
- Must implement the `operateOnPartition` method.
- Must be invoked with a PARTITION BY, which specifies how rows are reshuffled.
- “Sees” all the appropriate rows in a partition.
Requirements of a SQL/MR Function

- Must implement either `RowFunction` or `PartitionFunction`.
- Must have a single-argument constructor which takes a single `RuntimeContract` as a parameter.
- Class name must be all lowercase.
- Name of jar file must be the same as the SQL/MR function name.

Note: can also upload a `<functionname>.zip` file, containing multiple jars. The jar with the SQL/MR function must have same name as the function, but other jars can be included. Useful for including libraries.
The Constructor

```java
public tokenize(RuntimeContract contract) {
    ArrayList<ColumnDefinition> output = new ArrayList<ColumnDefinition>();

    outputColumns.add(
        new ColumnDefinition("word", SqlType.varchar()));
    outputColumns.add(
        new ColumnDefinition("count", SqlType.bigint()));

    contract.setOutputInfo(new OutputInfo(outputColumns));
    contract.complete();
}
```
The constructor can throw exceptions. If the exception is a subclass of `ClientVisibleException`, the user sees a descriptive message on the command line tool. Otherwise, they see a generic error message.

A full stack trace of the exception can be viewed via the AMC.
SELECT ...
FROM FunctionName(
    ON {tablename | (subquery)}
    [PARTITION BY ...]
    [ORDER BY ...]
    ARGCLAUSE1 (..., ...)
    MYCLAUSE (...)
    ...
);
Tip 1: CTAS

BEGIN;

\install tokeninze.jar

CREATE FACT TABLE counts (PARTITION KEY(word)) AS
SELECT word, sum(count)
FROM tokenize(ON documents)
GROUP BY word;
ORDER BY word;

END;
Tip 2: Use Transactions

BEGIN;

\install tokeninze.jar

CREATE FACT TABLE counts (PARTITION KEY(word)) AS
SELECT word, sum(count)
FROM tokenize(ON documents)
GROUP BY word;
ORDER BY word;

END;
Tip 3: PARTITION BY $c$

BEGIN;

\install exact_percentile.jar

SELECT *
FROM exact_percentile(
    ON source_data
    PARTITION BY 1
    PERCENTILE(25, 50, 75)
);

ABORT;
Tip 4: Using act

To connect to the cluster, use the command line tool **act**.

bash$ act -h <ip-address> -d <databasename> -U <username>

*Useful commands*

\d List all tables.
\d <table name> Show table details.
\dF List installed SQL/MR files.
\? More detailed help.
\timing Enable query timing.
BEGIN;
\install tokenize.py

SELECT word, sum(count)
FROM STREAM(
    ON documents
    SCRIPT('tokenize.py')
    OUTPUTS('word varchar', 'count int')
)
GROUP BY word
ORDER BY word;
ABORT;
movie_titles. Stores movie id, year, and titles.

training_set. Main training dataset. Stores (customerid, movieid, viewdate, and rating).

probe_set. A random sample of (customerid, movieid) pairs from the training set. Designed to be used for testing your classifier.

qualifying_set. A set of (customerid, movieid, viewdate) rows not in the training set. To enter the contest, submit your classifier’s ratings for these movies.
Both the probe and qualifying sets are ordered. The file you submit to Netflix needs to be in that same order. Therefore, the `probe_set` and `qualifying_set` tables have an extra `entryid` column.

See [www.netflixprize.com](http://www.netflixprize.com) for more details about the dataset and on entering the contest.