NanoDB+ Final Report

Angela Gong
Mike Qian
Kalpana Suraesh

June 7, 2012
CS 123, Spring 2012

1 General Overview

The goal of this project, spanning 10 weeks of the Spring 2012 term, was to implement the grouping and aggregation functionality in NanoDB. Since it did not take too long to implement grouping and aggregation, more functionality was added, such as the ability to use `HAVING` clauses, as well as functions and utilities such as `SHOW TABLES`.

The code was written in Java and uses the git version control system. The repository is located on the CMS cluster at `/cs/courses/cs123/sp2012/nanodb-grouping.git/`.

2 Architectural Overview

We will highlight important classes and functions that we have changed or implemented to support the new functionalities.

res/nanosql.g (*sqlparse*)

- Added parser support for `DESCRIBE`, `GROUP BY`, `HAVING`, `LIMIT`, and `SHOW TABLES`, as well as explicit support for aggregate function calls.
- Syntax for aggregate functions is hard-coded to ensure correct parsing of arguments and to avoid nondeterminism issues with regular functions.

functions/(Aggregate)?FunctionCall

- Enabled creation of function call objects during the query parsing phase that can later be used to evaluate a function against a set of one or more tuples.
- Detects attempts to evaluate nested aggregates, which are disallowed by the SQL standard.

functions/FunctionDirectory

- Completed support for function calls, including aggregates `MIN/MAX`, `SUM/AVG`, `STDDEV/VAR`, `COUNT/COUNT(*)`, and using `DISTINCT.`

plans/GroupingFactory

- Allows NanoDB to use either hashing or sorting to compute groups for evaluation of aggregates. All *GroupAggregateNode*s should be instantiated using this class, not directly.

plans/(Hashed|Sorted)?GroupAggregateNode

- *GroupAggregateNode* takes in a child node, a list of group by expressions, a list of select values, and an optional `HAVING` expression. It extracts the aggregate expressions, which can be retrieved with `getAggregateExprs()`.
- `getModifiedSelectValues()` returns the select values with all the aggregate function calls replaced by column values. `getModifiedHavingExpr()` does the same for the (optional) `HAVING` expression.
- *HashedGroupAggregateNode* organizes groups by hashing tuples against the grouping columns, and then calculates aggregate values for each group via calls to the `evaluate()` method of the appropriate *AggregateFunctionCall* objects.
- *SortedGroupAggregateNode* organizes groups by sorting all the tuples according to the grouping columns, and then calculates aggregate values for each group via calls to the `evaluate()` method of each aggregate.

qeval/DPJoinPlanner

- Modified plan generation to include support for grouping/aggregation, including the `HAVING` expression. Also allows composite expressions like `a + MIN(b)` and `MIN(a + b)` via the *GroupAggregateNode* classes.
- Since aggregates are initially represented by *AggregateFunctionCall* objects in the list of select values, a modified list with these expressions replaced by column values is retrieved from the grouping/aggregation plan-node.

expressions/Expression class hierarchy

- `retrieveAllExprs()` returns all subexpressions categorized by type.
- `retrieveExprsByType()` returns all subexpressions of a given type.
- `containsExpressionType()` checks whether the expression has any sub-expressions of the given type.
- `replaceAllInstances()` replaces all occurrences of a given expression with another expression.
- Modifications enable support for composite expressions involving aggregates and the `HAVING` expression.

PrettyTuplePrinter

- Added the `print()` function which prints all of the resulting tuples at once.
- Aligns columns (including right-aligning numerical data).
- Parameter `limit` in the constructor tells the printer to limit the amount of output displayed, and `offset` is the offset from which tuples will start being printed.

commands/DescribeCommand

- `addColInfo()` adds information about each field in the table to describe it, including its name, type, whether or not it allows null values, default values, etc.
- `execute()` gets column info from the schema and passes it to `addColInfo()`.

commands/ShowTablesCommand

- `checkExistence()` makes sure that the list of tables exist and creates it if not.
- `addTable()` and `removeTable()` adds and removes a row, respectively, from the table that contains all of the tables.

tests/TestAggregation, *TestGroupBy*, *TestHaving*, and *TestGroupingAndAggregation*

- Added tests cases for grouping and all types of aggregation with and without the `HAVING` clause.

3 Main Challenges

3.1 Parser Issues

In getting the parser to recognize both standard and aggregate function calls, we ran into some issues working with *FunctionCall* objects. The existing rule that recognized function syntax expected a *FunctionCall* return type, but since *AggregateFunctionCall*s comprise a different class of objects and do not simply extend the functionality of *FunctionCall*s, we had to refactor aggregate recognition into a new parser rule and explicitly specify the supported calls. While support for standard functions is specified within the *FunctionDirectory* class, aggregates generally have the same syntax as standard function calls and thus have to be hard-coded in order to avoid potential non-determinism warnings.

3.2 Grouping Nodes

Initial tests appeared to show that both the sorted and hashed implementations of the grouping/aggregation plan-node were working as intended. However, further digging seemed to indicate that `VARCHAR` columns were not being properly sorted into groups. This required a simple fix: we accidentally sorted using the wrong schema. Another problem we ran into while trying to hash tuples was that *TupleLiteral* did not implement the `equals()` or `hashCode()` methods, so there was no way to compare partial tuples outside of the *TupleComparator* class. Our temporary fix involved hashing the tuple's string value; once we updated *TupleLiteral*, we changed the hash implementation to use the tuple directly.

3.3 Implementing the `HAVING` Clause

For some time, we wanted to provide support for the `HAVING` expression in conjunction with grouping/aggregation, but we couldn't get it to work just as we couldn't yet provide support for composite select values involving aggregates such as `a + MIN(b)`. Eventually we modified the *Expression* class hierarchy to store a field that identifies each expression's type (*e.g.*, *AggregateFunctionCall*, *BooleanOperator*, *ColumnValue*, etc.), thus allowing us to continue referencing expression objects generically using the base *Expression* class. All classes that extend *Expression* can also be analyzed in order to extract all expressions of a given type. After this, we were able to compute aggregates regardless of where they appear in the select values or `HAVING` expression.

3.4 Tests

We were very confused for a while because our tests kept failing, until we realized that *TupleLiteral* had a constructor to create tuples with several `NULL` values, which we were misusing by thinking it would create a tuple with a single integer value. We also had some problems related to spacing in the `test_sql.props` file.

4 Checking Out and Building

4.1 Repository

Once git is installed on a machine, the repository can be checked out from the CMS cluster using the command:

```
git clone username@login.cms.caltech.edu:/cs/courses/cs123/sp2012/nanodb-grouping.git
```

The remote repository may be updated by first committing the changes to the local repository using `git commit`, and then pushing to the remote via `git push`.

4.2 Building and Tests

The source code for NanoDB+ is located in the `src/` directory, and can be automatically built by running `ant compile`. Running `ant clean` will delete the build directory so that all the files can be rebuilt from scratch.

Tests are run using the TestNG framework, and the files are located in the `tests/` directory. They can be run using the command `ant test`, and the results can be viewed at `build/results/index.html`.

4.3 Documentation

Documentation is created with Javadocs, and can be generated by running `ant javadoc` after the source code has been built. The result is located at `build/javadoc`. The javadoc index can be found at `build/javadoc/index.html`.

5 Features in Action

5.1 Starting up NanoDB+

Run NanoDB+ by typing into `sh nanodb`, `./nanodb`, or `nanodb` into a shell after building it using `ant compile`.

```
~shell>nanodb
Welcome to NanoDB.  Exit with EXIT or QUIT command.

CMD>
```

5.2 Grouping and Aggregation

Simple grouping can be done on columns.

```
CMD> SELECT str FROM table_1 GROUP BY str;
```

```
+------+
| STR  |
+------+
| asdf |
| blah |
| five |
| four |
| noky |
| okay |
| spam |
+------+
```

Aggregate functions can be selected as well as part of the query.

```
CMD> SELECT str, SUM(num) FROM table_1 GROUP BY str;
+------+-----------------+
| STR  | SUM(TABLE_1.NUM) |
+------+-----------------+
| asdf |          5723.5 |
| blah |           123.4 |
| five |           123.4 |
| four |            40.1 |
| noky |           358.5 |
| okay |          1341.0 |
| spam |            95.0 |
+------+-----------------+
```

More complicated aggregate expressions involving arithmetic operations can be done.

```
CMD> SELECT str, id + AVG(num) FROM table_1 GROUP BY str, id;
+------+------------------------------+
| STR  | TABLE_1.ID + AVG(TABLE_1.NUM) |
+------+------------------------------+
| asdf |                       5729.5 |
| blah |                        124.4 |
| five |                        129.4 |
| four |                         45.1 |
| noky |                         66.3 |
| noky |                        298.2 |
| okay |                         41.1 |
| okay |                       1237.6 |
| okay |                         70.3 |
| spam |                         70.3 |
| spam |                        102.7 |
+------+------------------------------+
```

The `HAVING` clause may be used to filter the results of the `GROUP BY` expression.

```
CMD> SELECT str, SUM(id) FROM table_1 GROUP BY str HAVING SUM(id) < 3;
+------+-----------------+
| STR  | SUM(TABLE_1.ID) |
+------+-----------------+
| blah |               1 |
+------+-----------------+
```

5.3 Functions

An example of a function that can be run is computing `SIN` of a column. Other functions can be run similarly.

```
CMD> SELECT SIN(a) FROM foo LIMIT 5;
+------------------------+
| SIN(FOO.A)             |
+------------------------+
|     0.5805375529616218 |
|   -0.04355002626945846 |
|    0.07966873759898874 |
|    -0.7392416009402618 |
|    -0.9939616415067287 |
+------------------------+
```

5.4 Utility Commands

5.4.1 SHOW TABLES

NanoDB+ has the ability to show all the tables in the database. This command can be seen in action by creating and dropping many tables, and seeing the changes in what `SHOW TABLES` displays.

Upon first running `SHOW TABLES` in a clean database, the following output should appear:

```
CMD> SHOW TABLES;
Empty set
```

A couple of tables can be created:

```
CMD> CREATE TABLE a ( a INTEGER );
Created table:  A
Table of tables initialized.
CMD> CREATE TABLE b ( b INTEGER );
Created table:  B
CMD> CREATE TABLE c ( c INTEGER );
Created table:  C
```

By running the `SHOW TABLES` command the tables can be seen.

```
CMD> SHOW TABLES;
+-----------------+
| Tables_in_NanoDB |
+-----------------+
| A               |
| B               |
| C               |
+-----------------+
```

If you drop a table, the list of tables will be updated accordingly.

```
CMD> DROP TABLE b;
CMD> SHOW TABLES;
+-----------------+
| Tables_in_NanoDB |
+-----------------+
| A               |
| C               |
+-----------------+
```

5.4.2 DESCRIBE Command
Using the `DESCRIBE` command, details about the parameters of the table are listed. This can be run using the `DESCRIBE <table_name>` command.

First a table must be created.

```
CMD> CREATE TABLE b (
   >    a INTEGER,
   >    b VARCHAR(30),
   >    c FLOAT
   > );
Created table:  B
```

`DESCRIBE <table_name>` can then display the table's fields.

```
CMD> DESCRIBE b;
+-------+-------------+
| Field | Type        |
+-------+-------------+
| A     | INTEGER     |
| B     | VARCHAR(30) |
| C     | FLOAT       |
+-------+-------------+
```

5.4.3 LIMIT Command

We can limit the amount of output that appears. For example we have a table with 1000 rows, but we only want to display the first 10 in the console.

We can see that there are 1000 rows in this table.

```
CMD> SELECT COUNT(*) FROM limit_demo;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
```

Now we can limit the amount of output that appears.

```
CMD> SELECT * FROM limit_demo LIMIT 5;
+------+-----------+
| A    | B         |
+------+-----------+
|  945 | lbps      |
| 8055 | jst       |
| 1263 | rgcgeaoel |
| 1003 | kizoebfgf |
| 7890 | c         |
+------+-----------+
```

We can also offset the amount of output that appears. For example, the command below will display 3 after an offset of 1 (remember that it is 0-indexed).

```
CMD> SELECT * FROM limit_demo LIMIT 1, 3;
+------+-----------+
| A    | B         |
+------+-----------+
| 1263 | rgcgeaoel |
| 1003 | kizoebfgf |
| 7890 | c         |
+------+-----------+
```