CS165 Project for Fall 2016

Overview

The goal of the project is to design and build a main-memory optimized column-store.

By the end of the project you will have designed, implemented, and evaluated several key elements of a modern data system and you will have experienced several design tradeoffs in the same way they are experienced in all major data industry labs.

This is a heavy but fun project! We will also point to several open research problems throughout the semester that may be studied on top of the class project and that you may decide to take on as a research project if you like.

The project runs through the whole semester. There are a total of 5 milestones with specific expected software deliverables, which will be accompanied with a design document. The deliverables will be tested using predefined automated tests for functionality and, as extra credit, performance. We will provide these tests before-hand so you can test your system as you develop it.

We will provide you with starting code that implements the basic client-server functionality, i.e., anything that has to do with how a server communicates with the client over a socket. In addition, we will provide a parser for the domain specific language. In this way your focus will be on designing and building the core of the database server, i.e., the essential core data processing algorithms and data structures of a database system. Whenever applicable we will let you know if there are existing libraries that are OK to use.

The project includes work on storage methods such as arrays for plain storage and B-trees for clustered or secondary indexes. It also includes work on access methods such as fast scans that are optimized for modern hardware and can utilize properties such as CPU prefetching and multi-cores. We expect you to provide main-memory optimized designs and implementation for several core operators of a db system: select, fetch, join, aggregations and basic math. In addition, the project includes work on updates and efficient execution of workloads that include multiple concurrent queries using scan sharing.

Putting everything together you will have a data system that gets as input an intermediate language, i.e., a language used by db optimizers to describe a physical query plan. The system will be able to run complete select-project-join queries and it will be at least an order of magnitude faster than traditional row-store database systems in analytical workloads.

Logistics

Implementation language: The implementation language for the project is C. This is because a low-level language is required to design and build efficient systems as we need to have full control of how we utilize memory and computational resources. (Part of our bonus tasks include using alternative languages.)

Compiler: gcc 4.7.2 will be used to compile your code. You should not rely on any features or functionality that have been added since this version. gcc is generally backward compatible so if you are using an older version of gcc you should be fine but you should confirm this by running your code on the testing server.

Libraries: Before using any library check with the instructor and the TFs. In general, we will not allow any library that has to do with data structures. The reason is that one of the main objectives of the course is to learn about the trade-offs in designing these data structures/algorithms yourself. By controlling the data structures, you have control over every last bit in your system's data and you can minimize the amount of data your system moves around (which is the primary bottleneck).

IDE: You may use any IDE and development environment that you are familiar with. As long as your project compiles to our machine, everything is OK. We propose using Vim or Emacs and do not generally provide support for any other specific IDE.

Evaluation: Individual deliverables should pass the provided tests. However, you will not be judged only on how well your system works; it should be clear that you have designed and implemented the whole system, i.e., you should be able to perform changes on-the-fly, explain design details, etc.

At the end of the semester each student will have a 1-hour session with the instructor and another 1-hour session with the TFs where they will demonstrate the system and answer design questions about the current design and about supporting alternative functionality. [Tip: *From past experience we found that frequent participation in office hours, brainstorming sessions and sections implies that the instructor and the TFs are very well aware of your system and your progress which makes the final evaluation a mere formality for these cases.*]

Collaboration Policy: The project is an individual project: the final deliverable should be personal, you must write from scratch all the code of your system and all documentation and reports. Discussing the design and implementation problems with other students is allowed and encouraged! We will do so in the class as well and during office hours, sections and brainstorming sessions.

Late days policy: We allow for 1000 late days or until Harvard requires us to upload your grade! The more input you give us, the more we can help you learn. On the project website and in the project description you can find a detailed time-schedule that we propose you follow. With the exception of the midway check-in (which is a hard deadline), the rest is a “suggested schedule” that will allow you to spread the work throughout the semester and to have sufficient time for each milestone based on the complexity and the work required at each phase of the project. This is an involved project that requires commitment through the entire semester and cannot be done in 2-3 weeks at the end. Not submitting the project milestones on time will have no side- effects on your grade but at the same time, we will not be able to provide you with any feedback on your progress until we have your design documents and your code.

Note: Experience says that every year a number of students cannot handle the freedom to self-pace, and end up significantly deviating from the schedule. We will send you frequent reminders but you should know that deviating from the schedule by more than a couple of weeks will most likely mean that you will not be able to finish the whole project by the end of the semester (unless you are an experienced systems student).

Midway Check-in: The goal here is to demonstrate that you are having decent progress and mainly to avoid falling behind. ByOctober 31st midnight (hard deadline) each student should 1) deliver a design document that describes the intended design for the first three milestones (5%) and 2) have implemented a project that passes at least the first three tests of the first milestone in the automated testing infrastructure (5%). A template of the expected design document will be provided early in the semester.

Bonus points for extra tasks: We will regularly assign extra tasks or you can come up with your own extra tasks for the various components of the project. With these extra tasks you gain extra points.

Best projects: The best 3 overall projects will gain additional extra points. "Best" is defined in terms of elegant system design, code quality, system efficiency and documentation.

Leaderboard: We will have a running competition and an anonymous leaderboard infrastructure so you can continuously test your system against the rest of the class.

Grading: Each of the 5 milestones account for 18% of the total project grade. The midway point check-in counts for the remaining 10% of the project grade. Each of the bonus tasks counts for an extra 0.25% of the total class grade.

Starting Code and API

We have created a repository with handout code that you can use to get started on your project. The goal of providing this code is twofold. First, in this repository we provide the parts of your system that are not directly related to data systems (such as client/server communication). Second, we give you a starting implementation of various parts of the database system that can help you with your own design. You are free to build upon the given code or to start over with your own architecture.

You will find the repository on code.seas. Please clone the repository into your own working repository. We will notify you if/when the base code changes (bug fixes or added functionality), but you may also want to pull from the master periodically.

To add the repository, do the following:

1. Login to code.seas.
2. Search for the handout code repository: cs165-2016-base.
3. Click "Clone repository".
4. Choose a new name if you want, then click "Clone repository".

This will bring a copy of the base code into your personal account. You can now clone, pull, and push to this copy. Please make sure to add read permission for cs165-staff to both your project and the repository so we can view your code submissions. If you are unfamiliar with working with git, here is a nice summary: **http://cs61.seas.harvard.edu/wiki/2016/Git**.

Milestones

MILESTONE 1: BASIC COLUMN-STORE - SUGGESTED COMPLETION DATE: October 2

In this milestone, the goal is to design and implement the basic functionality of a column-store with the ability to run single-table queries. The first part of this milestone is about the storage and organization of relational data in the form of columns, one per attribute. This requires the creation and persistence of a database catalog containing metadata about the tables in your database as well as the creation and maintenance of files to store the data contained in these tables. While the data in your database should be persistent, it does not need to be fault tolerant. In addition, your database will only need to support columns storing integer data.

The next step is to implement a series of scan-based operators on top of the columns to support queries with select, project, aggregation and math operators. Then using these operators you will synthesize single-table query plans such as *SELECT max(A) FROM R where B< 20 and C > 30* that work with a bulk processing data flow. It should be noted that even when columns store only integer data, the results of aggregation statements (such as AVG) may produce non-integer results.

Desired Functionality:

After implementing this milestone we expect that your system will be able to support loading and retrieving data from a simple column-store. In particular the system after Milestone 1 should be able to *create* a database, *load* a file, *insert* new rows, *select* data from a column, *fetch* data from a column, and calculate *aggregates* (max, min, avg, sum, add, sub). For example, after creating a database, a table, and the corresponding columns:

```
create(db,"awesomebase") -- create db with name "awesomebase"
create(tbl,"grades",awesomebase,6) -- create table "grades" with 6 columns in
the "awesomebase"
create(col,"project",awesomebase.grades) -- create column 1 with name
"project"
create(col,"midterm1",awesomebase.grades) -- create column 2 with name
"midterm1"
create(col,"midterm2",awesomebase.grades) -- create column 3 with name
"midterm2"
create(col,"class",awesomebase.grades) -- create column 4 with name "class"
create(col,"quizzes",awesomebase.grades) -- create column 5 with name
"quizzes"
create(col,"student_id",awesomebase.grades) -- create column 6 with name
"student_id"
```

The system should be able to insert a few values:

```
relational_insert(awesomebase.grades,107,80,75,95,93,1)
relational_insert(awesomebase.grades,92,75,82,90,85,2)
relational_insert(awesomebase.grades,110,95,90,100,95,3)
relational_insert(awesomebase.grades,88,70,75,85,95,4)
```

In order to eventually, be able to select and fetch to implement a very simple query:

```
a_plus=select(awesomebase.grades.project,90,100) -- Find the students (rows)
with project grade between 90 and 100
ids_of_students_with_top_project=fetch(awesomebase.grades.student_id,a_plus) -
- Find the student id of the students
```

To complete this milestone your database will need to implement a variable pool and database catalog. The variable pool is transient and keeps track of the state of client-server communication. This includes keeping track of named intermediate results. For instance, “a_plus” in the previous example is an intermediate result specific to the client that executed the query. These variables are not persistent and should be released on client shutdown. In contrast, the database catalog is a persistent structure that tells your database how to locate tables, columns, and their metadata. This information must be persisted on shutdown and recreated when the database starts up again.

Bonus Tasks:

1. Add support for variable length data.
2. Implement vectorized processing plans and provide a performance comparison with the bulk processing ones.
3. Implement this milestone in an alternative language of your choice and provide a performance comparison and analysis against the C implementation. We propose you use Go.

MILESTONE 2: INDEXING - SUGGESTED COMPLETION DATE: OCTOBER 23

The goal is to add indexing support to boost select operators. Your column-store should support memory-optimized B-tree indexes and sorted columns in both clustered and unclustered form. In the first case, all columns of a table follow the sort order of a leading column(s). We expect query plans to differentiate between the two cases and properly exploit their physical layout properties. The end goal is to be able to run single-table queries (as in the first milestone) but use the new index-based select operators instead. Your column-store should support multiple clustered indices as well as multiple unclustered indices. To support multiple clustered indices, your database must organize and keep track of several copies of your base data. To make the creation of these clustered indices easier, we place several restrictions on them:

1. All clustered indices will be declared before the insertion of data to a table.
2. The first declared unclustered index will be the principal copy of the data. Only this copy of the data will support primary indices.

Desired Functionality:

After the second milestone, the system should be able to create columns organized using a B-Tree. For example we may want an index for column 6 of the previous example in order to locate easily each student based on their id. In this case, we should be able to give:

```
create(idx,awesomebase.grades.student_id,btree)
```

> **TIP|** Be careful of random access during tuple reconstruction after an unclustered index-based select.

We expect your B-tree to be tuned for main-memory, i.e., to minimize cache misses by properly tuning design parameters such as the fan-out of the tree based on the underlying hardware as well as any other design parameter that may affect performance. You are allowed to move away from the basic B-tree design.

Your deliverable should include a performance analysis (i.e., graphs and discussion) that shows the performance difference between your various select methods (alternative tree designs, scans, and binary search) based on various parameters that affect performance such as selectivity and number of tuples.

Bonus Tasks:

1. Add support for zone-maps and provide performance comparison with scans and indexes.
2. Add support to declare indices after the creation of data.
3. Implement this milestone in an alternative language of your choice and provide a performance comparison and analysis against the C implementation. We propose you use Go.

MILESTONE 3: FAST SCANS: SCAN SHARING & MULTI-CORES - SUGGESTED COMPLETION DATE: NOVEMBER 6

The third milestone is about making your scans fast. You will be adding support for scan sharing to minimize data movement and a multi-core NUMA-aware design to fully utilize parallelization opportunities of modern CPUs. Specifically, we will add scan-sharing capabilities to the scan-based select operator.

Your system should be able to run N>>1 queries in parallel by reading data only once for the select operator. You may assume that all queries arrive concurrently. The end goal is to be able to run simple single-table queries (as in the first milestone) but use the new select operators that are capable of scan sharing.

To introduce more opportunities for shared scans, we introduce batching operators. The client can declare a batch of queries to execute and then tell the server to execute them all at once. The server then must coordinate with the client when it is done with this batch. In this batching operation, it can be assumed that no print commands will be executed. That is, no commands to the database will need relational results.

The end result of this milestone should be a linear scale up with the number of concurrent queries and number of cores. Your deliverable should include a performance report (graphs and discussion) to demonstrate that you can achieve such a performance boost. This report should discuss your results with respect to the various parameters that affect shared scan performance such as the number of queries and the number of cores.

We also expect you to answer the following question: How many concurrent queries can your shared scan handle? Is there an upper limit? If yes, why?

Desired Functionality:

After Milestone 3, the main effort is in optimizing scanning and fetching, hence, the relevane calls should be much faster. The following call should run now much faster.

```
batch_queries()
a_plus=select(awesomebase.grades.project,90,100) -- Find the students (rows)
with project grade between 90 and 100
a=select(awesomebase.grades.project,80,90) -- Find the students (rows) with
project grade between 80 and 90
super_awesome_peeps=select(awesomebase.grades.project,95,105)
ids_of_students_with_top_project=fetch(awesomebase.grades.student_id,a_plus) -
- Find the student id of the a_plus students
batch_execute() -- The three selects should run as a shared scan
```

Bonus Tasks:

1. Utilize scan sharing for queries from multiple clients
2. Utilize scan sharing for additional operators.
3. Utilize multi-cores for the B-tree index.
4. Implement this milestone in an alternative language of your choice and provide a performance comparison and analysis against the C implementation. We propose you use Go.

MILESTONE 4: JOINS - SUGGESTED COMPLETION DATE: NOVEMBER 20

For the fourth milestone you will be adding join support to your system. We expect you to support cache-conscious hash joins and nested-loop join that utilize all cores of the underlying hardware.

In addition you need to support query plans that use these new join operators, e.g.:

```
SELECT max(R.d), min(S.g)
FROM R, S
WHERE
    R.a = S.a
 AND R.c >= 1
 AND R.c =< 9
 AND S.f >= 31
 AND S.f =< 99
```

Your plans should use late materialization and bulk processing.

TIP| Make sure you minimize random access during fetch operators after a join.

The deliverable for this milestone includes a performance report of nested loops vs hash joins.

Desired Functionality:

Milestone 4 introduces joins, a key relational operation. The result would be that the system would support joins, and be able to join two tables. For example, after creating two tables and populating them with tuples, the user should be able to join them:

```
positions1=select(awesomebase.cs165.project_score,100,null) -- select
positions where project score >= 100 in cs165
positions2=select(awesomebase.cs265.project_score,100,null) -- select
positions where project score >= 100 in cs265
values1=fetch(awesomebase.cs165.student_id,positions1)
values2=fetch(awesomebase.cs265.student_id,positions2)
r1,r2=join(positions1,values1,positions2,values2,hash) -- positions of
students who have project score >= 100 in both classes
```

Bonus Tasks:

1. Add support for sort-merge joins and provide a performance comparison with hash joins.
2. Download an open-source row-store DBMS such as PostgreSQL and an open-source column-store DBMS such as MonetDB and provide a comparison with your system on this milestone's tests.
3. Implement this milestone in an alternative language of your choice and provide a performance comparison and analysis against the C implementation. We propose you use Go.

MILESTONE 5: UPDATES - SUGGESTED COMPLETION DATE: DECEMBER 11

For the final milestone we will be adding update support. We expect you to support inserts, deletes, and updates on relational tables. Any changes to data should be persistent, i.e., they should not get lost if we restart the server.

The goal is to add update and locking support. Your column-store should be able to handle concurrent updates. In this milestone you should be able to run workloads where read queries interleave with updates (inserts, deletes or actual updates). Multiple queries should be able to work in parallel, updating/reading the same data.

Updating base data requires changing data in each copy of the data. This means keeping metadata around in some form such that updates made on one copy of the data get propagated to the other copies of the data! Come see the TFs early and often on this front if unsure about the best way to achieve this. One suggested way is to keep a table which maps row numbers in your primary copy to each of the other copies. Then given positions for one copy of the data, the other positions can be found using a join on this table.

Desired Functionality:

In the final milestone general updates should be supported. In addition to inserts, relational deletes and relational updates should be implemented.

```
low_project = select(awesomebase.grades.project,0,10) -- Find the rows with project less than 10
relational_delete(awesomebase.grades,low_project) -- clearly this is a mistake!!
-- or update
update(awesomebase.grades.project,low_project,85) -- change value of column project of table grades
```

Bonus Tasks:

1. Implement this milestone in an alternative language of your choice and provide a performance comparison and analysis against the C implementation. We propose you use Go.

MIDWAY EVALUATION: October 31

EVALUATION: DECEMBER 12-15 (Hard Deadline)

CS165 Domain Specific Language

Here you can find the syntax of the CS165 Domain Specific Language (165L). In a full system design, this language would be used by the optimizer to pass instructions (i.e., query plans) to the execution engine after parsing and optimizing incoming SQL queries. In this project you only work at the lower level, i.e., the storage and execution engine, and you will use 165L to write query plans to test your engine. As 165L is the interface between your data system and our test files, supporting the exact 165L syntax is mandatory.

A few details about things you will encounter in the rest of the language description:

1. Keywords are unqualified text and symbols. For example: *create*, *tbl*, *col* etc. (These are static words that you can use to parse the instructions. They will appear in the same order and location in the string).
2. Items that appear in brackets are required but indicate good opportunities for extensions, or relate to one of the extra features found in the project description. For example, 'create(idx,<col_name>,[btree])' means that you must support creating at least B-tree indexes, but you may want to also support additional indexes like zone maps or hash maps.
3. Variables appear in-between angle brackets. They are strings that appear in 165L and are either identifiers like the name of a table or are labels that the system must carry through the execution of the commands (this will become more clear through the examples).
4. End of line indicates next instruction (but your design can buffer or parse multiple lines at a time as you see fit).
5. Comments are marked with '--' and continue until the end of the line.

CREATING NEW DATABASE OBJECTS

```
create(<object_type>,<object_name>,<parameter1>,<parameter2>,...)
```

The create function creates new structures in the system. The possible structures are *databases*, *tables*, *columns*, and *indexes*. It does not return anything. Below you can see all possible instances.

```
create(db,<db_name>)
create(tbl,<t_name>,<db_name>,<col_cnt>)
create(col,<col_name>,<tbl_var>)
create(idx,<col_name>,[btree, sorted], [clustered, unclustered])
```

Usage

```
create(db,"awesomebase") -- create db with name "awesomebase"
create(tbl,"grades",awesomebase,6) -- create table "grades" with 6 columns in
the "awesomebase"
create(col,"project",awesomebase.grades,unsorted) -- create column 1 with name
"project"
create(col,"midterm1",awesomebase.grades,unsorted) -- create column 2 with
```

```
name "midterm1"
create(col,"midterm2",awesomebase.grades,unsorted) -- create column 3 with
name "midterm2"
create(col,"class",awesomebase.grades,unsorted) -- create column 4 with name
"class"
create(col,"quizzes",awesomebase.grades,unsorted) -- create column 5 with name
"quizzes"
create(col,"student_id",awesomebase.grades,unsorted) -- create column 6 with
name "student_id"
```

SQL Example

```
CREATE DATABASE awesomebase

CREATE TABLE grades (grades int, project int, midterm1 int, midterm2 int,
class int, quizzes int, student_id int)
```

In the create table statement, the first value of a parameter is the column name and the second parameter is its type. VARCHAR(n), BINARY(n), BIGINT, and TIMESTAMP are examples of other SQL data types.

LOADING

`load(<filename>)`

This function loads values from a file. Both absolute and relative paths should be supported. The columns within the file are assigned names that correspond to already created database objects.

Parameters

<filename>: The name of the file to load the database from.

Return Value

None.

Usage

```
load("/path/to/myfile.txt")
-- or relative path
load("./data/myfile.txt")
```

File Format

Input data will be provided as ASCII-encoded CSV files. For example:

```
foo.t1.a,foo.t1.b
10,-23
-22,910
```

This file would insert two rows into columns 'a' and 'b' of table 't1' in database 'foo'.

SQL Example

There is no direct correlate in SQL to the load command. That being said, almost all vendors have commands to load a file into a table. The MySQL version would be:

```
LOAD DATA INFILE myfile.txt
```

INSERTING ROWS INTO A TABLE

The system will support relational, that is, row-wise (one row at a time) inserts:

```
relational_insert(<tbl_var>,[INT1],[INT2],...)
```

Internally, the insert will have to be translated into a series of columnar inserts:

```
insert(<col_var>,[INT])
```

Parameters

<col_var>: A fully qualified column name.

<tbl_var>: A fully qualified table name.

INT/INTk: The value to be inserted (32 bit signed).

Return Value

None.

Usage

```
relational_insert(awesomebase.grades,107,80,75,95,93,1)
```

SQL Example

There are two different insert statements in SQL. In the first statement below, the column names are omitted and the values are inserted into the columns of the table in the order those columns were declared in table creation. In the second statement, column names are included and the values in the insert statement are put in the corresponding given column. The two statements below perform the same action.

```
INSERT INTO grades VALUES (107,80,75,95,93,1)
```

```
INSERT INTO grades (midterm1, project, midterm2, class, quizzes, student_id)
VALUES (80,107,75,95,93,1)
```

SELECTING VALUES

There are two kinds of select commands.

Select from within a column:

```
<vec_pos>=select(<col_var>,<low>,<high>)
```

Parameters

<col_name>: A fully qualified column name or an intermediate vector of values

<low>: The lowest qualifying value in the range.

<high>: The exclusive upper bound.

null: specifies an infinite upper or lower bound.

Select from pre-selected positions of a vector of values:

```
<vec_pos>=select(<posn_vec>,<col_var>,<low>,<high>)
```

Parameters

<posn_vec>: A vector of positions

<col_var>: A vector of values.

<low>: The lowest qualifying value in the range.

<high>: The exclusive upper bound.

null: specifies an infinite upper or lower bound.

Return Value

<vec_pos>: A vector of qualifying positions.

Usage

```
-- select (type 1)
pos_1=select(awesomebase.grades.project,90,100) -- Find the rows with project
score between 90 and 99
pos_2=select(awesomebase.grades.project,90,null) -- Find the rows with project
greater --or equal to 90
-- select (type 2)
pos_1_values=fetch(awesomebase.grades.midterm,pos_1) -- Fetch the values at
pos_1 from the midterm column
pos_1_temp=select(pos_1,pos_1_values,95,100) -- Within the fetched values,
find the rows with midterm between 95 and 99
```

SQL Example

```
SELECT student_id FROM grades WHERE midterm1 > 90 AND midterm2 > 90
```

In the statement above, we might select on midterm1 using the first select, then select on midterm2 using the second type of select.

FETCHING VALUES

This function collects the values from a column at given positions.

```
<vec_val>=fetch(<col_var>,<vec_pos>)
```

Parameters

<col_var>: A fully qualified column name.

<vec_pos>: A vector of positions that qualify (as returned by select or join).

Return Value

<vec_val>: A vector of qualifying values.

Usage

```
a_plus=select(awesomebase.grades.project,100,null) -- Find the rows with
project greater or equal to 100
-- used here
ids_of_top_students=fetch(awesomebase.grades.student_id,a_plus) -- Return
column 6 (student id) of the said rows (students)
```

DELETING ROWS

Similarly to the insert operation, deleting values and rows is built in two steps. The relational_delete operation will internally issue multiple separate column deletes.

```
relational_delete(<tbl_var>,<vec_pos>)
```

```
delete(<col_var>,<vec_pos>)
```

Parameters

<col_var>: A fully qualified column name.

<tbl_var>: A fully qualified table name.

<vec_pos>: A vector of positions.

Return Value

None.

Usage

```
low_project=select(awesomebase.grades.project,0,10) -- Find the rows with
project less than 10
-- used here
relational_delete(awesomebase.grades,low_project) -- Clearly this is a
mistake!!
```

SQL Example

```
DELETE FROM grades WHERE midterm1 < 40 AND midterm2 < 40
```

JOINING COLUMNS

This function performs a join between two inputs, given both the values and respective positions of each input.

```
<vec_pos1_out>,<vec_pos2_out>=hashjoin(<vec_val1>,<vec_pos1>,<vec_val2>,<vec_p
os2>, <type>)
```

Parameters

<vec_val_1>: The vector of values 1.

<vec_pos_1>: The vector of positions 1.

<vec_val_2>: The vector of values 2.

<vec_pos_2>: The vector of positions 2.

<type>: The type of join to execute. An example of type is hash or nested loop.

NOTE: There is no explicit indication which is the smaller relation. why we care will become apparent when we discuss joins.

Return Value

<vec_pos1_out>,<vec_pos2_out>: The concatenation of the positions in each input table of the resulting join.

Usage

```
positions1=select(awesomebase.cs165.project_score,100,null) -- select
positions where project score >= 100 in cs165

positions2=select(awesomebase.cs265.project_score,100,null) -- select
positions where project score >= 100 in cs265

values1=fetch(awesomebase.cs165.student_id,positions1)

values2=fetch(awesomebase.cs265.student_id,positions2)

r1,r2=join(positions1,values1,positions2,values2, hash) -- positions of
students who have project score >= 100 in both classes

student_ids=fetch(awesomebase.cs165.student_id, r1)

print(student_ids)
```

SQL Example

```
SELECT student_id FROM cs165_grades JOIN cs265_grades
WHERE cs165_grades.project > 100
AND cs165_grades.project > 100
AND cs165_grades.student_id = cs265_grades.student_id
```

MIN AGGREGATE

There are two kinds of min aggregate commands.

`<min_val>=min(<vec_val>)`

The first min aggregation signature returns the minimum value of the values held in <vec_val>.

Parameters

<vec_val>: A vector of values to search for the min OR a fully qualified name.

Return Value

<min_val>: The minimum value of the input <vec_val>.

`<min_pos>,<min_val>=min(<vec_pos>,<vec_val>)`

The second min aggregation signature returns the minimum value and the corresponding position(s) (as held in <vec_pos>) from the values in <vec_val>.

Parameters

<vec_pos>: A vector of positions corresponding to the values in <vec_val>.

<vec_val>: A vector of values to search for the min OR a fully qualified name.

Note: When null is specified as the first input of the function, it returns the position of the min from the <vec_val> array.

Return Value

<min_pos>: The position (as defined in <vec_pos>) of the min.

<min_val>: The minimum value of the input <vec_val>.

Usage

```
positions1=select(awesomebase.grades.project,100,null) -- select students with
project more than or equal to 100
values1=fetch(awesomebase.grades.midterm1,positions1)
-- used here
min1=min(values1) -- the lowest midterm1 grade for students who got 100 or
more in their project
```

SQL Example

```
SELECT min(midterm1) FROM grades WHERE project >= 100
```

MAX AGGREGATE

There are two kinds of max aggregate commands.

`<max_val>=max(<vec_val>)`

The first max aggregation signature returns the maximum value of the values held in <vec_val>.

Parameters

<vec_val>: A vector of values to search for the max OR a fully qualified name.

Return Value

<max_val>: The maximum value of the input <vec_val>.

`<max_pos>,<max_vals>=max(<vec_pos>,<vec_val>)`

The second max aggregation signature returns the maximum value and the corresponding position(s) (as held in <vec_pos>) from the values in <vec_val>.

Parameters

<vec_pos>: A vector of positions corresponding to the values in <vec_val>.

<vec_val>: A vector of values to search for the max OR a fully qualified name.

Note: When null is specified as the first input of the function, it returns the position of the max from the <vec_val> array.

Return Value

<max_pos>: The position (as defined in <vec_pos>) of the max.

<max_val>: The maximum value of the input <vec_val>.

Usage

```
positions1=select(awesomebase.grades.midterm1,null,90) -- select students with
midterm less than 90
values1=fetch(awesomebase.grades.project,positions1)
-- used here
max1=max(values1) -- get the maximum project grade for students with midterm
less than 90
```

SQL Example

```
SELECT max(project) FROM grades WHERE midterm1 < 90
```

SUM AGGREGATE

`<scl_val>=avg(<vec_val>)`

This is the aggregation function sum. It returns the sum of the values in <vec_val>.

Parameters

<vec_val>: A vector of values.

Return Value

<scl_val>: The scalar value of the sum.

Usage

```
positions1=select(awesomebase.grades.project,100,null) -- select students with
project more than or equal to 100
values1=fetch(awesomebase.grades.quizzes,positions1)
-- used here
sum_quizzes=sum(values1) -- get the sum of the quizzes grade for students with
project more than or equal to 100
```

SQL Example

```
SELECT SUM(quizzes) FROM grades WHERE project >= 100
```

AVERAGE AGGREGATE

`<scl_val>=avg(<vec_val>)`

This is the aggregation function average. It returns the arithmetic mean of the values in <vec_val>.

Parameters

<vec_val>: A vector of values.

Return Value

<scl_val>: The scalar value of the average.

Usage

```
positions1=select(awesomebase.grades.project,100,null) -- select students with
project more than or equal to 100
values1=fetch(awesomebase.grades.quizzes,positions1)
-- used here
avg_quizzes=avg(values1) -- get the average quizzes grade for students with
project more than or equal to 100
```

SQL Example

```
SELECT AVG(quizzes) FROM grades WHERE project >= 100
```

ADDING TWO VECTORS

`<vec_val>=add(<vec_val1>,<vec_val2>)`

This function adds two vectors of values.

Parameters

<vec_val1>: The vector of values 1.

<vec_val2>: The vector of values 2.

Return Value

<vec_val>: A vector of values equal to the component-wise addition of the two inputs.

Usage

```
midterms=add(awesomebase.grades.midterm1,awesomebase.grades.midterm2)
```

SQL Example

```
SELECT midterm1 + midterm2 FROM grades
```

SUBTRACTING TWO VECTORS

```
<vec_val>=sub(<vec_val1>,<vec_val2>)
```

This function subtracts two vectors of values.

Parameters

<vec_val1>: The vector of values 1.

<vec_val2>: The vector of values 2.

Return Value

<vec_val>: A vector of values equal to the component-wise addition of the two inputs.

Usage

```
-- used here
score=sub(awesomebase.grades.project,awesomebase.grades.penalty)
```

SQL Example

```
SELECT AVG(project - penalty) FROM grades
```

UPDATING VALUES

This function updates values from a column at given positions with a given value.

`update(<col_var>,<vec_pos>,[INT])`

Parameters

<col_var>: A variable that indicates the column to update.

<vec_pos>: A vector of positions.

INT: The new value.

Return Value

None.

Usage

```
project_to_update=select(awesomebase.grades.project,0,100) -- ...it should
obviously be over 100!
-- used here
update(awesomebase.grades.project,project_to_update,113)
```

SQL Example

```
UPDATE grades SET midterm1 = 100 WHERE midterm2 = 100
```

PRINTING RESULTS

print(<vec_val1>,...)

The print command prints one or more vector in a tabular format.

Parameters

<vec_val1>: One or more vectors of values to be combined and printed.

Return Value

None.

Usage

```
-- used here
print(awesomebase.grades.project,awesomebase.grades.quizzes) -- print project
grades and quiz grades
--OR--

pos_high_project=select(awesomebase.grades.project,80,null) -- select project
more than or equal to 80
val_project=fetch(awesomebase.grades.project,pos1) -- fetch project grades
val_studentid=fetch(awesomebase.grades.student_id,pos1) -- fetch student id
val_quizzes=fetch(awesomebase.grades.quizzes,pos1) -- fetch quiz grades
-- used here
print(val_studentid,val_project,val_quizzes) -- print student_id, project
grades and quiz grades for projects more than or equal to 80
```

Then, the result should be:

```
107,1
92,2
110,3
88,4
```

SQL Example

This instruction is used to print out the results of a query. As such, this command is used in every query in a database which returns values.

Batching Commands

Batching consists of two commands. The first command, batch_queries, tells the server to hold the execution of the subsequent requests. The second command, batch_execute, then tells the server to execute these queries.

`batch_queries()`

`batch_execute()`

Return Value

batch_queries: none.

batch_execute: No explicit return value, but the server must work out with the client when it is done sending results of the batched queries.

Usage

```
batch_queries()
a_plus=select(awesomebase.grades.project,90,100) -- Find the students (rows)
with project grade between 90 and 100
a=select(awesomebase.grades.project,80,90) -- Find the students (rows) with
project grade between 80 and 90
super_awesome_peeps=select(awesomebase.grades.project,95,105)
ids_of_students_with_top_project=fetch(awesomebase.grades.student_id,a_plus) -
- Find the student id of the a_plus students
batch_execute() -- The three selects should run as a shared scan
```

SQL Example

There is no batching command in the SQL syntax. However, almost all commercial databases have a command to submit a batch of queries.

SHUTTING DOWN

This command shuts down the server.

`shutdown`

Return Value

None.

Design Document and Feedback

For every milestone, once you have a basic design that you think you want to try and before you start implementing it, create a 1-2 page PDF with the design and give it to us for feedback (through Canvas). We will reply within 1-2 days at most. Even better, come with your design during office hours or sections so we can review it live together. Your design does not have to be perfect; the idea is that we can give you early feedback, help you with your design decisions, and point to possible optimizations.

Note, that there is hard deadline to deliver a design document for the first 3 milestones as part of the midway check in on October 31st. After that the design document is optional for the rest of the milestones but strongly encouraged.

Practicing SQL and Column-store Plan

A great way to practice your SQL skills is to go to the state-of-the-art benchmark for analytical workloads (TPCH) and study the queries. TPCH is a very popular ad-hoc decision support benchmark used both by academia and industry to test data systems.

You can install any db system you prefer (e.g., MonetDB, MySQL, PostgreSQL or a combination of them).

And use the database generator (dbgen) of the TPCH data in order to create raw files which you can then bulk load.

In the specification of the TPCH you can find:

1. the TPCH schema in page 13
2. all 22 TPCH queries in English and SQL starting from page 29 (subsections 2.4.1.1, 2.4.2.1, 2.4.3.1, ... , 2.4.22.1 have the English description and 2.4.1.1, 2.4.2.2, 2.4.3.2, ... , 2.4.22.2 have the SQL of them).

You can use these queries, which are typically an excellent and challenging exercise for thinking how to "convert" English to SQL as well as formulate more queries of your own on top of this schema.

In addition, it is a good idea to practice with column-store plans so that you can get a better understanding of concepts such as late tuple reconstructions and how data flows as queries get more complex. This will help you a lot with your project. Use MonetDB and prefix your SQL queries with the keyword “explain”. What you will see then is the column-store plan (equivalent of our DSL language). (Use MonetDB in read-only mode to get a simpler plan that does not consider on-the-fly merging of pending updates.)

What is a Successful Project ?

A successful project *passes all the predefined tests* we provide and the students successfully *pass the final face-to-face evaluation*.

A *successful final evaluation* is one where the student is able:

1. to fully explain every detail of the design, and

2. to propose efficient designs for new functionality on the spot.

A project will get extra points for going the extra mile to provide solutions that are efficient and are elegantly implemented and designed. Efficiency is judged by comparing against other systems. Elegance is judged by the course staff. Participating in office hours and extra sessions guarantees that you get feedback about your design often.

A project also gets extra points for doing some of the bonus tasks we provide or bonus tasks that the student comes up with.