AlexDB: A Relational DBMS for Statistical Analysis

1st Alexander Chenot
Computer Science
Colorado School of Mines
Golden, USA
achenot@mines.edu

I. Introduction

The primary function of a database, perhaps unsurprisingly, is to store data; however, what use does data storage have if we never actually do anything with the data? One strategy for taking action with data is to move it to an external client which works well in a lot of cases, but in other cases, it may be convenient to perform data processing and storage on the same system to skip the intermediate step of moving data and to elide the need for maintaining two separate systems. Consider the case of a data scientist who is performing a small-scale experiment where they need to collect some data samples and compute statistics about individual samples and all collected data as a whole; it would be convenient for this data scientist to have a system that can both hold the data samples and automatically compute nontrivial statistics without any needed intervention beyond some initial configuration work.

Alex Database (AlexDB) is a relational database management system (DBMS) that combines the tasks of storing data and performing nontrivial statistical analysis on that data into a single easy-to-use system. The key contributions of AlexDB include SQLScript, which is a dialect of SQL that embeds a functional programming language into queries, an ability to perform advanced statistical analysis via calculated columns and aggregate calculations, and a variety of columnar compression techniques that ensure data is stored efficiently in memory.

II. Related Work

A. DuckDB

DuckDB [2] is an in-memory and column-oriented database focused on data analytics, much like AlexDB and DuckDB offers many features similar to those proposed by AlexDB. DuckDB only provides a fixed default set of row aggregate functions and if a user wants to write their own, they have to write an extension to DuckDB which can be a confusing and time-consuming process, especially to those without the necessary technical knowledge. In contrast, all aggregates in AlexDB are custom, and defining custom aggregates using SQLScript is a straightforward and intuitive process.

B. Gorilla

Gorilla [3] is an in-memory and column-oriented timeseries database that performs very effective data compression that directly insprired AlexDB. As a whole, AlexDB is much more open-ended and flexible than Gorilla, since Gorilla is a timeseries database with a fixed schema.

C. C-Store

C-Store [4] is a column-oriented database with compression techniques and the ability to compute aggregates. C-Store only provides a fixed set of aggregates, while AlexDB provides support for custom aggregates via SQLScript.

D. PostgreSQL

PostgreSQL [1] is very popular and complex DBMS that is highly configurable. While PostgreSQL can do everything that AlexDB does in terms of custom aggregates, calculated fields, and compression methods, actually configuring those things to work requires a significant depth of knowledge about the PostgreSQL DBMS. Compared to PostgreSQL, AlexDB is very easy to spin up an instance of on commodity hardware or include in a separate program as a library, and the only thing a user needs to know to configure an instance of AlexDB is SQLScript, which is a relatively simple language.

III. Architecture

AlexDB is formulated as a Rust crate (similar to a library or module) which can be included in other applications. The main entry point into AlexDB is a `Database` object which the user interacts with by giving queries in the form of strings. The Database object interface is shown in Table I.

TABLE I
Database Object Interface

Method	Inputs	Outputs
execute	query: String	QueryResult

`QueryResult` is an Enum type with possible values shown in II.

On a call to `execute`, data follows the pipeline shown in Figure 1. Each item in the pipeline is explained in detail in the following sections.

An example of AlexDB as a REPL is included in the AlexDB source code [5].

TABLE II
QUERYRESULT ENUM INTERFACE

Value	Description
Table(Table)	Table result, returned by SELECT queries.
Value(Val)	Script language value result.
Error(String)	Error result.
Success(String)	Generic success message.
Exit	Return value that indicates an exit request.

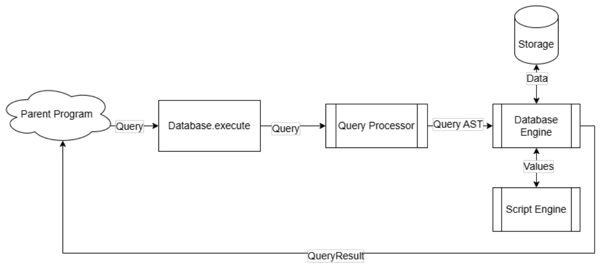

Fig. 1. AlexDB System Architecture

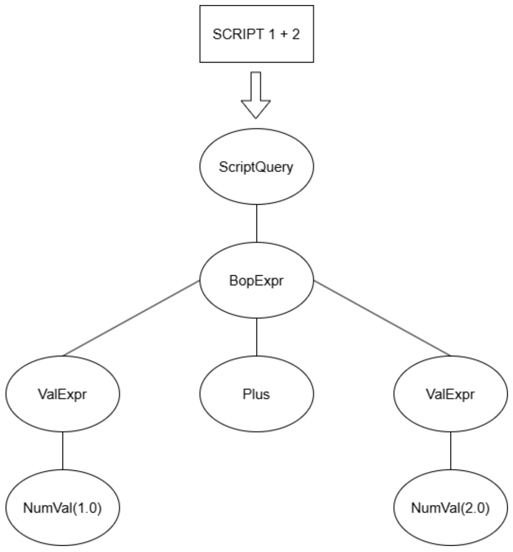

Fig. 2. Example Abstract Syntax Tree

IV. QUERY PROCESSOR

It is the query processor's responsibility to turn every query represented as a string into an equivalent representation as an abstract syntax tree (AST), which the database engine uses to decide how to take action.

A. Lexer

The query lexer inputs a query as a string and outputs a sequence of tokens that represents the input query. AlexDB's lexer uses lazy evaluation by only producing the next token in a sequence when requested to by the parser, which saves memory since not all tokens that represent a sequence are stored in memory at once. Example lexer outputs are shown in Table III

TABLE III
LEXER OUTPUT EXAMPLES

Input String	Output Sequence
`SELECT * FROM table`	`[ SELECT_KW; TIMES; FROM_KW; IDENT(table) ]`
`SCRIPT 1 + 2`	`[ SCRIPT_KW; NUMBER(1.0); PLUS; NUMBER(2.0) ]`

B. Parser

The query parser inputs a sequence of tokens from the lexer and outputs an AST that represents the input query. AlexDB's parser uses an LL(2) recursive descent method with zero backtracking. All methods of the parser are LL(1) except for a method which looks ahead two tokens to check for an assignment operator ($\cdots = \ldots$), which makes the entire parser LL(2). An example output of the parser is shown in Figure 2. The grammar for the script language is described in the appropriate section, and the grammar for queries is omitted because it is very trivial.

V. QUERY LANGUAGE

The query language is the user's only point of interaction with AlexDB, and closely mimics SQL because SQL uses natural language so it is easy to write and users who are already familiar with SQL will have an easier experience acclimating to AlexDB.

A. Schema Creation

AlexDB provides the `CREATE TABLE` query from SQL to create a table schema. The user can optionaly specify which compression method they would like to use on each field, or let it default to `none`. The syntax for the `CREATE TABLE` command is shown in Figure 3. The type of a field can be one of $\{num, str, bool\}$ where num is a 64-bit floating point number.

Fig. 3. `CREATE TABLE` Syntax

```
CREATE TABLE
    name
    (
        field type [compression],
        field type [compression],
        ...
    )
```

B. Data Insertion and Extraction

In AlexDB, the user can either insert data into a table row-by-row or by importing from a CSV file. The syntax for inserting row-by-row follows a similar syntax from SQL and is shown in Figure 4. When inserting row-by-row, all script language `<expr>`'s are coerced into the data type for the field they belong to. When importing from a CSV file, column headers are used to map CSV columns to AlexDB table columns; any extra CSV columns are ignored and any missing columns are filled in with `NULL` values. The syntax for importing a CSV file into a table is shown in Figure 5.

Fig. 4. `INSERT INTO` Syntax

```
INSERT INTO
    table
    [(field, field, ...)]
    VALUES (<expr>, <expr>, ...)
```

Fig. 5. `IMPORT CSV` Syntax

```
IMPORT CSV 'path/to/csv.csv' INTO table
```

The data for AlexDB is stored entirely in memory, however the user may save the data stored in a table to disk with the `EXPORT CSV` query. Saving tables as CSV files is useful for data persistence and sharing. All exported CSV files include column headers to make re-importing into AlexDB trivial. The syntax for exporting data to a CSV file is shown in Figure 6.

Fig. 6. `EXPORT CSV` Syntax

```
EXPORT CSV 'path/to/csv.csv' FROM table
```

C. Global Constants

AlexDB allows users to register global constants that are available to any script language `<expr>` database-wide. Global constants allow for users to make shorter and more intuitive queries, and can be any available script language value, not just values that can be stored in tables. The syntax for creating global constants is available in Figure 7

Fig. 7. `CREATE CONST` Syntax

```
CREATE CONST name = <expr>
```

D. Calculated Columns

AlexDB users may use SQLScript to create advanced calculated columns in table schemas. The script language `expr` has access to all regular table fields, as well as all calculated column values that were registered before the current column (i.e., columns that appear "earlier" in the schema). Each time a calculated column is created, it is added to the end of the table schema. The syntax for creating a calculated column is shown in Figure 8.

E. Aggregates, Computations, and the "Mean Problem"

Aggregates are at the heart of AlexDB, and are registered into a table schema with the `CREATE AGGREGATE` query, shown in Figure 9. Aggregates are re-computed each time a row is inserted into a table and follow the `fold` paradigm from functional programming, which means that the next aggregate value is determined using only the current aggregate value, stored in a special variable called `current`, and all

Fig. 8. `CREATE COLUMN` Syntax

```
CREATE COLUMN
    (type [compression])
    name = <expr> INTO table
```

field values from the newly inserted row including calculated fields. The user may optionally supply an `INIT` parameter to the query, which initializes the aggregate using the first row inserted into the table; if the user does not supply an `INIT` parameter, then a `NULL` value is stored in `current` before the first row is inserted.

Fig. 9. `CREATE AGGREGATE` Syntax

```
CREATE AGGREGATE
    name = <expr>
    [INIT <expr>]
    INTO table
```

It is important that aggregates cannot see each other as they are being computed, since that would break the `fold` paradigm and potentially cause strange behavior, since there isn't a definitive schema for aggregates as there is for calculated columns. If aggregates are only able to see their current value and the most recent inserted row, then some statistics that need additional information are impossible to calculate. Consider, for example, the mean statistic. The mean is most famously calculated as $\frac{sum(values)}{count(values)}$, however without consider composite data types, it is impossible to define an aggregate that keeps track of both the sum of a field and the number of rows inserted into a table, which I have coined the "mean problem." To solve the mean problem, AlexDB includes computations, which act as a method of combining aggregate values. The syntax for computations is available in Figure 10. Computations solve the mean problem because a user can define two separate aggregates, one to count the number of rows inserted into the table and the other to sum a specific field, then the user can use a computation to divide the sum aggregate by the count aggregate to compute the mean of a column.

Fig. 10. `CREATE COMP` Syntax

```
CREATE COMP
    name = <expr>
    INTO table
```

It is worth circling back to the claim that the mean problem only exists without composite data types like tuples, because SQLScript has tuples as an available data type for aggregates. Using tuples, a user can define a single aggregate that stores both row count and column sum, and use the formula $mean = \frac{current \times count + field)}{count+1}$ to update the mean for a field, thus rendering the mean problem moot. Even with composite data types, computations are still useful because it is more intuitive to calculate many statistics, like the mean, with multiple aggregates, and computations can automatically select specific tuple fields from aggregates using tuples, which makes querying a little bit easier.

Finally, the syntax for querying both aggregates and comptuations is found in Figure 11.

Fig. 11. `SELECT AGGREGATE | COMP` Syntax

```
SELECT AGGREGATE | COMP
    name
    FROM table
```

F. Row Queries

AlexDB parially implements the `SELECT` query from SQL to allow users to query rows stored in a table. In addition to table fields, the `WHERE` clause `<expr>` has access to all table aggregations and computations. AlexDB's `SELECT` syntax is available in Figure 12.

Fig. 12. `SELECT` Syntax

```
SELECT
    * | field, field, ...
    FROM
    table
    [WHERE <expr>]
    [ORDER BY field [ASC | DESC]]
    [LIMIT <expr>]
    [EXPORT CSV 'path/to/csv.csv']
```

G. Script Query

AlexDB provides a query type that allows the user to manually compute a script language expression, optionally including aggregations and computations from a table; this query type is useful for testing or one-off computations. The syntax for the `SCRIPT` query is available in Figure 13.

Fig. 13. `SCRIPT` Syntax

```
SCRIPT
    <expr>
    [FROM table]
```

H. Compression

In addition to specifying a compression method for each column upon schema creation, AlexDB offers the ability to re-compress columns using the `COMPRESS` query; this query allows for re-compression on a column-by-column basis or bulk re-compression. The syntax for the `COMPRESS` query is available in Figure 14.

VI. Script Language

The script language is the *script* part of SQLScript and it is a simple but powerful functional programming language; this language was heavily inspired by JavaScript and can be trivially converted to a subset of JavaScript code.

Fig. 14. `COMPRESS` Syntax

```
COMPRESS
    table
    (field, field, ...)
    (strategy, strategy, ...) | strategy
```

A. Grammar

The context-free grammar for the script language is available in Table IV. Please note that this grammar is reduced in the sense that it is not LL(2) and does not consider operator precedence; the actual grammar is factored such that it is LL(2) and includes eight levels of precedence that are separated by separate `<Expr>` production rules.

TABLE IV
Script Language Grammar

⟨Start⟩	→	⟨Expr⟩ $
⟨Expr⟩	→	”fun” ⟨Identlist⟩ ”->” ⟨Expr⟩
	|	”if” ⟨Expr⟩ ”then” ⟨Expr⟩ ”else” ⟨Expr⟩
	|	⟨Expr⟩ ”(” ⟨ExprList⟩ ”)”
	|	⟨Uop⟩ ⟨Expr⟩
	|	⟨Expr⟩ ⟨Bop⟩ ⟨Expr⟩
	|	⟨Value⟩
⟨Value⟩	→	⟨Ident⟩
	|	⟨Number⟩
	|	⟨String⟩
	|	”null”
	|	”undefined”
	|	”(” ⟨Expr⟩ ”)”
	|	”[” ⟨ExprList⟩ ”]”
	|	”{” ⟨StmtList⟩ ”}”
⟨Number⟩	→	[0-9]+(\.[0-9]+)?
⟨String⟩	→	’[^’]*’
⟨Ident⟩	→	[a-zA-Z][a-zA-Z0-9_]*
⟨IdentList⟩	→	⟨Identlist’⟩
	|	λ
⟨IdentList’⟩	→	⟨Ident⟩ ”,” ⟨Identlist’⟩
	|	⟨Ident⟩
⟨ExprList⟩	→	⟨Exprlist’⟩
	|	λ
⟨ExprList’⟩	→	⟨Expr⟩ ”,” ⟨Exprlist’⟩
	|	⟨Expr⟩
⟨StmtList⟩	→	⟨Ident⟩ ”=” ⟨Expr⟩ ”;” ⟨StmtList⟩
	|	⟨Expr⟩

B. Functionality

Functional was the best paradigm for the script language because functional programming tends to be more declarative, which means that you can do a lot with less; most users writing queries do not want to consider low-level programming details, therefore the more declarative the better, to a certain extent. Furthermore, every SQLScript expression evaluates to a single value, which is a desirable property because the query engine expects a value every time it runs script language code. Like many functional programming languages, the script language is entirely immutable and implements lexical function scope, which means that functions use their define-time variable environment, rather than their call-time variable environment.

The only aspect of the script language that deviates from the ”pure functional” paradigm is that functions can access variables that were defined *after* the function was defined; for example, the expression `f = fun -> x; x = 10; f()` is a valid expression. This deviation from pure functionality is useful because many popular and familiar languages like JavaScript operate in this way, and it allows for functions to be recursive without explicitly needing to name them.

C. Engine

The script engine is a virtual machine that exists independently of the query engine. Whenever the query engine needs to evaluate script language code, it spins up a new instance of the script engine, loads all necessary values into the engine's environment, then calls on the engine to evaluate the code. For example, for each `SELECT` query with a `WHERE` clause, the query engine determines which whether or not to keep a row by loading each field of the row, all table aggregates, and all table computations into a query environment, then running the `WHERE` clause code using that environment and checking the boolean value of the output.

The script engine uses a big-step evaluation model to evaluate an input AST, which means that expressions are evaluated all at once rather than iteratively until they reach a fixed point. The big-step evaluation model is less performant than the small-step evaluation model because it tends to build very large call stacks, however it is much easier and more intuitive to implement.

The script engine maintains a virtual call stack to handle variable scope. When entering a new block, the script engine pushes a new stack frame to the virtual call stack; each stack frame maps variable identifiers to values. For each `ident = expr` present in a block, the script engine adds a new mapping from `ident` to the value that `expr` evaluates to onto the topmost stack frame. When seaching for the value tied to an identifier, the script engine searches from the top of the stack down as to get the most recent value of that identifier. Upon exiting a block, the script engine simply pops the topmost stack frame off the virtual call stack. Figure 15 contains an example of how the call stack changes over the lifetime of a program.

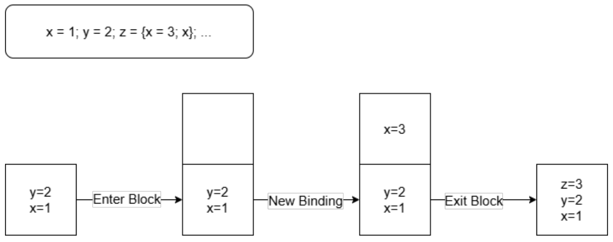

Fig. 15. Call Stack Example

Functions use the call stack to maintain lexical scope by taking a snapshot of their define-time environment, and storing that snapshot in a closure. An environment snapshot is a single stack frame that contains the most recent values for every variable in the environment; an example is available in Figure 16. When is a function is called, the snapshot stored in its closure is added to the top of the virtual call stack, which means that the function's define-time environment will be the first choice for variable values within the function body. Once a function body is finished evaluating, the function's snapshot is popped of the virtual call stack.

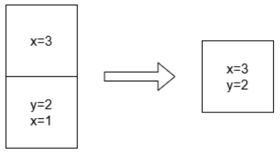

Fig. 16. Snapshot Example

D. Blocks

Blocks are environments that allow for the creation of variables. A block looks like a series of variable assignments followed by a final output value, surrounded by curly braces. For example, {`x = ...; y = ...; ...`}.

E. Tuples

A tuple is a composite data type that stores multiple values together. A tuple looks like a series of expressions surrounded by square brackets. For example, `[x, y, z, ...]`. Tuples are zero-indexed and accessed by the dot operator like so: `[x, y, z, ...].0`.

F. Functions

Functions are defined with the `fun` keyword and can include zero or more parameters. The function body is simply an expression whose evaluated value the function returns. For example, `fun x, y, z, ... -> ...` or `fun -> ...`. As with many programming languages, function calls are performed with parenthesis like so: `(fun x, y, ... -> ...)(a, b, ...)`.

G. Operators

SQLScript provides a rich set of unary and binary operators to operate on data. Table V shows all unary operators, and Table VI shows all binary operators. Only the binary table shows precedence, but unary operators have a higher precedence (tighter binding) than all binary operators except the dot operator. All binary operators with the same precedence are right-associative.

TABLE V
PREFIX UNARY OPERATORS

Operator	Description
−	Numerical Negation
!	Logical NOT
?	Boolean Type Conversion
&	String Type Conversion
+	Number Type Conversion
ˆ	Numerical Ceiling
_	Numerical Floor

TABLE VI
BINARY OPERATORS

Operator	Description	Precedence Level
&&	Logical AND	1
||	Logical OR	1
$>$	Strictly Greater	2
$\geq$	Greater	2
$<$	Strictly Less	2
$\leq$	Less	2
$==$	Loose Equality	2
$===$	Strict Equality	2
$+$	Addition	3
$-$	Subtraction	3
$*$	Multiplication	4
/	Division	4
%	Modulo	4
.	Tuple Access	5

VII. Data Storage

A. Columns

In AlexDB, columns are an abstract data type that implements the `Column` interface. The primary functions of the `Column` interface are to insert data into the column and to make the column act as an iterator that can be looped through to get each individual value stored in the column. The iterator interface is very convenient because when getting values from compressed columns, the compressed columns can intelligently reconstruct values only where the iterator is currently located, rather than needing to completely decompress when a value access is necessary.

B. Tables

The primary functionality of tables is to hold a collection of columns. Much like columns, tables also act as iterators by calling the iterator of each of its individual columns to return an entire row for each iteration. In addition to columns, tables also store the formulas and current values for aggregates and computations.

C. Database

The database object provides the `execute` interface discussed previously, and holds a collection of tables and global constants. The database object is most obviously used as a singleton object, however users can maintain multiple separate instances of AlexDB by maintaining multiple instances of the database object.

VIII. Data Compression

Data compression is done automatically by columns that implement the `Column` interface.

A. No Compression

No compression is the most straightforward kind of compression. This kind of compression is available for $\{str, num\}$ and operates as a vector that stores a sequence of a single data type.

B. Run Length Encoding

Run length encoding is available for $\{str, num\}$ and is based on Type 1 encoding from [4]. Run length encoding stores a sequence of pairs of $(value, count)$ that compress contiguous sequences of identical values. Run length encoding works best for data with few unique values where similar values appear close to each other. An example of run length encoding is shown in Figure VII.

TABLE VII
RUN LENGTH ENCODING EXAMPLE

TABLE VIII
ORIGINAL DATA

1
1
1
2
2
$NULL$

TABLE IX
COMPRESSED DATA

$(1, 3)$
$(2, 2)$
$(NULL, 1)$

C. Bitmap Encoding

Bitmap encoding is available for $\{str, num\}$ and is based on Type 2 encoding from [4]. For each unique value, bitmap encoding stores a pair of $(value, map)$ where map is a bitmap that indicates which rows the value appears in. It is convenient to store `NULL` values in bitmap encoding; a `NULL` value is present if each bitmap is false for a certain row, and the entire column is `NULL` if there are no pairs and the length is nonzero. An example of bitmap encoding is shown in Figure X

TABLE X
BITMAP ENCODING EXAMPLE

TABLE XI
ORIGINAL DATA

1
1
1
2
2
$NULL$

TABLE XII
COMPRESSED DATA

$(1, 111000)$
$(2, 000110)$

D. XOR Compression

AlexDB implements XOR compression from [3] for $\{num\}$, however the original algorithm needed to be modified to fit `NULL` values and arbitrary 64-bit floating points. XOR compression is effective when values aren't unique but neighboring values are close together. The modified algorithm is available in Algorithm 1

E. Boolean Compression

Because boolean values are a single bit, they can be stored very efficiently in a bitmap. Algorithm 2 is AlexDB's implementation of a boolean bitmap that accounts for `NULL` values.

Algorithm 1 XOR Compression

```
1) The first non-NULL value is stored with no compression preceeded by a '1' bit. Leading NULL values are indicated by '0' bits.
2) If the inserted value is NULL, insert a '0' bit
3) If the inserted value is not NULL, insert a '1' bit
    a) If XOR with the previous non-NULL value is zero, insert a '0' bit
    b) If the XOR is non-zero, insert a '1' bit
        i) If the block of meaningful bits falls within the previous block of meaningful bits, store a '0' followed by only the meaningful XORed value
        ii) Otherwise, store a '1' bit followed by the length of the number of leading zeros in the next 6 bits, then store the length of the meaningful XOR in the next 7 bits, then store the meaningful bits of the XOR value.
```

Algorithm 2 Boolean Compression

```
1) If the inserted value is NULL, insert a '0' bit
2) If the inserted value is non-NULL, insert a '1' bit followed by the inserted value as a bit
```

IX. Compression Results

To test the efficiency of AlexDB's compression schemes, I propose two datasets that represent different use cases, each dataset has a million samples. To test memory usage, I imported each of these datasets (stored as a CSV) into AlexDB, and used Valgrind and Massif to accurately determine the memory usage of an AlexDB instance running in a Linux environment.

A. Integer Data

The first dataset involves random integers between 0 and 20; this dataset represents a use case where you have few unique values. I tested this dataset both when these values are unsorted, a sample of which is shown in Figure 17 and when they are sorted, a sample of which is shown in Figure 18.

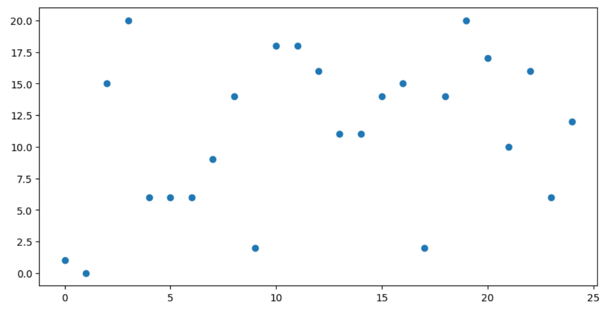

Fig. 17. Unsorted Integer Data, 25 Points

The memory testing results of this dataset are available in Figure 19.

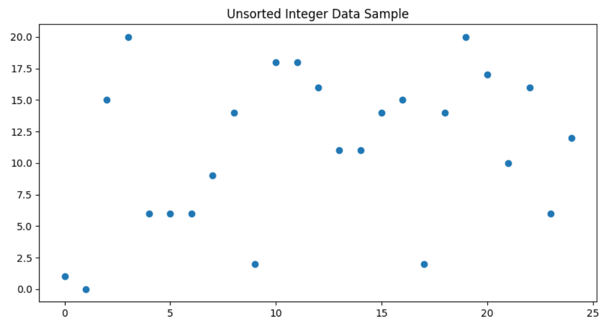

Fig. 18. Sorted Integer Data, 500,000 Points

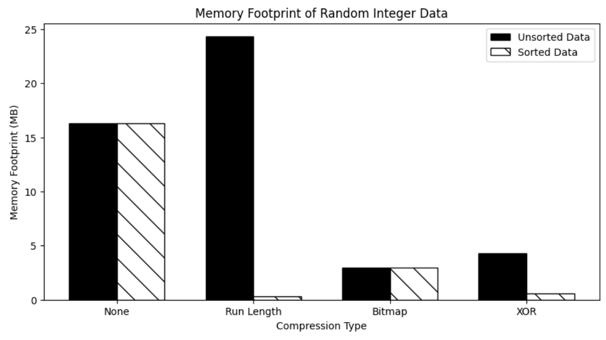

Fig. 19. Integer Dataset Memory Results

These results are fantastic but not surprising. Run length encoding works best overall on the sorted data, but worst overall on the unsorted data because of all the extra overhead. Bitmap encoding works the same regardless of the dataset being sorted or unsorted, and works the best overall in the unsorted case. XOR compression is always good but is outperformed in both cases by methods that are better suited to few unique values.

B. Floating-Point Data

The second dataset involves a sine wave with amplitude 10; this dataset represents a use case where data comes from some underlying distribution. A sample of this dataset is shown in Figure 20.

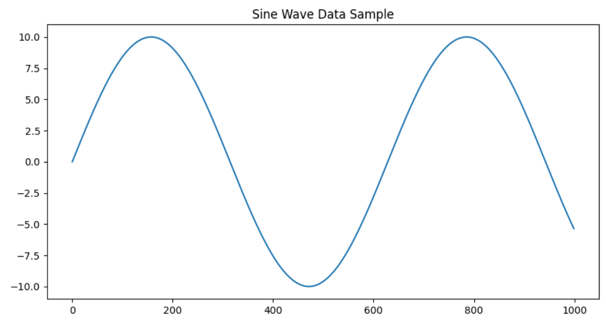

Fig. 20. Sine Dataset Sample, 1,000 Points

The memory testing results of this dataset are available in Figure 21.

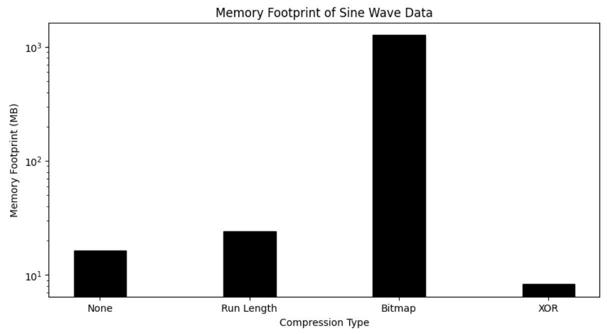

Fig. 21. Sine Dataset Memory Results

Much like the first dataset, the results are great but unsurprising. Bitmap encoding took over a gigabyte of memory as compared to the roughtly 25 megabyte baseline of no compression, which means that bitmap encoding is a very bad choice for this type of data. Run length encoding also increased over no compression, but with less catastrophic results than bitmap encoding. As expected, XOR compression provides a large improvement over no compression because XOR compression was designed for this type of data.

X. Future Work

AlexDB already includes a lot of functionality, however there is still a lot of opportunity for improvement.

1) Implement table joins, AlexDB does not currently support this and joining is an important component of a relational database system.
2) Ensure ACID database properties. AlexDB is single-threaded so consistency and independence are trivially guaranteed. Only durability for table data is guaranteed because users can save table data to a CSV, durability for any script language based item like aggregates, computations, global constants, and calculated column formulas is not available. Furthermore, some operations like re-compression can return an error in the middle of processing, leaving the operation only partially done, so atomicity is also not guaranteed.
3) Distribute AlexDB, either as a multi-node or multi-threaded application. Parallel processing could significantly speed up operations, especially for large amounts of data.

XI. Conclusion

AlexDB is a relational database system that provides a dialect of SQL called SQLScript which embeds a functional programming language in SQL queries. Furthermore, AlexDB offers a variety of columnar compression methods to save valuable memory space. AlexDB is targeted at statistical analysis, and offers advanced custom aggregates and calculated table fields. Some existing database systems already achieve the functional goals of AlexDB, however AlexDB makes the process of creating custom aggregates and calculated fields very up-front and intuitive. AlexDB is formulated as a library to be included as part of other programs, and users can use AlexDB by simply instantiating a `Database` object and calling `Database.execute` using queries as strings. Queries are first processed by the lexer and parser, then handed off to the database engine, which interacts with the script engine and the storage to fulfill a query. AlexDB offers a wide variety of query types, and the script language includes many helpful operators and data types to assist with the formulation of advanced calculations. Finally, I proved that the compression methods are effective by compressing data from multiple distributions.

References

[1] PostgreSQL, https://www.postgresql.org/

[2] M. Raasveldt and H. Mühleisen. DuckDB: an Embeddable Analytical Database. 2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, https://doi.org/10.1145/3299869.3320212

[3] T. Pelkonen et. al. Gorilla: A Fast, Scalable, In-Memory Time Series Database. VLDB. Proceedings of the VLDB Endowment, Volume 8, Number 12. https://www.vldb.org/pvldb/vol8/p1816-teller.pdf

[4] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2018. C-store: a column-oriented DBMS. Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker. Association for Computing Machinery and Morgan Claypool, 491–518. https://doi.org/10.1145/3226595.3226638

[5] Alex Chenot. 2024. AlexDB. https://github.com/chenota/alexdb