Algebraic Query Language (AQL): An Implementation of Relational Algebra

Written by:

Marc Kidwell Pestana

Edited by:

Kevin David Mitchell

and Arvin Donner

Algebraisk, LLC

www.algebraisk.com

scient@algebraisk.com

Table of Contents

1 *Abstract*

Algebraic Query Language (AQL) was developed at the Jet Propulsion Laboratory to track, study, and preserve engineering data products. It is a close implementation of E. F. Codd's relational algebra [9]. AQL is a relational DBMS based neither upon a natural language syntax nor upon strict data integrity constraints. We contrast AQL with SQL and present a formal definition of the former. We demonstrate that AQL is both a computational and a query language. To implement relational algebra in AQL, it was necessary to invent a datatype management system. This system is called the Automatic Type Database (ATD) and is described.

2 *Note to the reader*

For readers who are familiar with relational databases, or for those who only want to read about the AQL language, please proceed to the section titled “The AQL Database Model”. For those interested in the history of E. F. Codd and relational theory, or would prefer a historical explanation of AQL, please continue reading from here. In either case, some discussion of the background of E. F. Codd, SQL, and the relational model is necessary to properly explain AQL.

To keep the discussion from getting lost in technical details, wherever possible, we have tried to begin from an elementary staring point and then proceed to more advanced topics. Also, rigorous mathematical definitions are not provided accept by links and references to other sources.

3 *AQL and the relational database*

3.1 *Introduction*

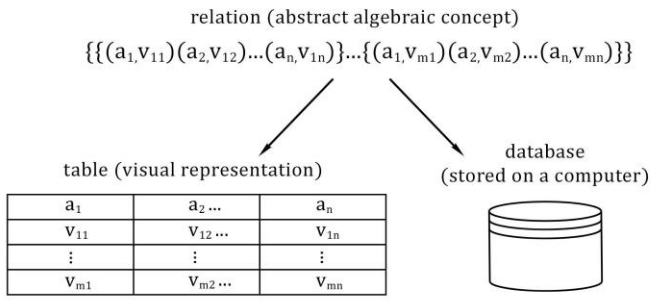
This paper examines notable aspects of Algebraic Query Language (AQL) and the database management system based upon it (the AQL DBMS or hereafter ADBMS). We compare AQL with SQL since both are relational languages that share common historical roots and yet take dramatically different approaches to language and database design. Marc Kidwell Pestana developed AQL for the GPS Earth Observatory (GEO) group at the Jet Propulsion Laboratory. AQL has been under development since 2010. It is fully operational and used in science and engineering research at JPL. AQL is the intellectual property of the California Institute of Technology in Pasadena.

Although relational databases and SQL are considered synonymous, both AQL and SQL have their roots in the relational model. By exploring the history of the relational model, we will discover the point of departure which differentiates them. This survey is crucial to understanding what AQL is and why it was developed.

3.2 Background

E. F. Codd was the inventor and evangelist of the *relational database model* [3][4]. In the early 1970's he specified the relational model for database design while working for IBM in San Jose, California [6]. Codd proposed a universal data format and methodology by which databases were to be manipulated with the CRUD architecture (Create, Read, Update, Delete)).Codd's relational model is a mathematical model for database design. The *relation* was the mathematical notion Codd used as his starting point and the basis for his universal data format. Using the relation, Codd invented models for both languages and programs designed to access relational databases under CRUD. The languages he invented are called *relational algebra* and *predicate calculus* [7].

The relation


relation (abstract algebraic concept)
$\{\{(a_1, v_{11})(a_2, v_{12})\ldots(a_n, v_{1n})\}\ldots\{(a_1, v_{m1})(a_2, v_{m2})\ldots(a_n, v_{mn})\}\}$
table (visual representation)
database (stored on a computer)

a_1	$a_2 \ldots$	a_n
v_{11}	$v_{12} \ldots$	v_{1n}
$\vdots$	$\vdots$	$\vdots$
v_{m1}	$v_{m2} \ldots$	v_{mn}

Figure 1. shows the set theoretic notation of a relation, its more common visualization as a table, and its representation in database design. The terms table and relation are often used interchangeably.

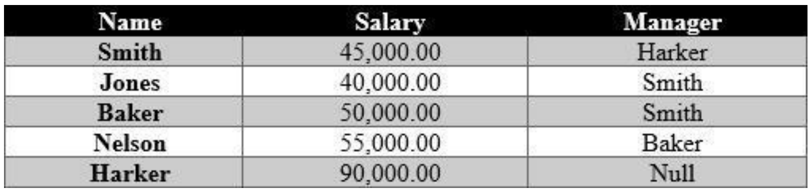
The relation (see *Figure 1*) is at the heart of the relational model. It is defined mathematically as a specialized kind of set (The formal definition for those interested can be found here [4, p. 4-5]) or on the Algebraisk website.) A relation combines data from one or more related sources into a homogeneous mathematical structure. By way of a very literal and informal example, consider the following statement assumed as a fact "Tom is the son of John." Here we are talking about both the mathematical structure and the abstract relationship between father and son. Using the table as a visualization, this statement takes the following form:

Son	**Father**
Tom	*John*

Of course, we can extend this relation with more factual statements of the same kind: "Mark is the son of Luke," and "Fred is the son of Mathew". Each of these statements become additional rows or records in the table. Inclusion of these statements in our example yields:

Son	**Father**
Tom	*John*
Marc	*Luke*
Fred	*Mathew*

The headers "Son" and "Father" are called the *attributes* of the relation. The attributes of a relation serve many functions: they label and/or describe the domain of the entries that lie in the column beneath them, they provide a unique path to each row, and they serve as a means of combining two relations together that share common attributes.



Name	Salary	Manager
Smith	45,000.00	Harker
Jones	40,000.00	Smith
Baker	50,000.00	Smith
Nelson	55,000.00	Baker
Harker	90,000.00	Null

Figure 2. Employee Table of Chamberlain and Boyce

Relations are often used to represent even more complicated statements of fact with up to hundreds of attributes relating hundreds of individual datum. Consider a less complicated relation whose attributes are "Employee", "Salary", and "Manager". Here we are defining a relation via its attributes. Assuming the fact that "Jones makes $40,000.00 a year and works for Smith", we group together the datum that fit these attributes to build a record in the table (see second row in Figure 2). Records, or rows, are also called tuples in formal mathematics and these terms are used interchangeably.

Codd went further in his development of his relational model. He exploited the fact that relations, defined as abstract sets, are naturally endowed with the set

operations of union, intersection, complementation, and difference. Codd modified these set operations by adding to them the property of closure. By enforcing closure, the set operations were changed into what he called relational operations.
Closure means that relations when combined by relational operations, produce relations. And if the operations are executed on properly defined relations, their composition creates new logical inferences. Codd used the relational operations to develop *data sublanguages* or query languages [7][8]. Relational algebra and predicate calculus were his two primary examples.

In summary, using the relation, Codd wrote a database model that specified the relation as the criteria for a universal data format and a data sublanguage for data access. The relation exhibits what he called *data independence* [2]. The relation exhibits data independence because it is self-contained, has a fixed homogeneous structure, and relies only on mathematical concepts that are independent of any machine representation or organization. He further recognized that it could be used to store assertions about abstract ideas. The relation could be used to package ideas into factual statements, or units of knowledge. Also, the relational operations could be used to form new inferences by combining these assertions.

Thus, in Codd's relational model, the relation functions on two levels. First, it provides a uniform, self-contained mathematical structure that can store and retrieve data, and can participate in relational operations. Second, it describes the factual relationships that exist between the various attributes of the data it stores and can be used to derive inferences from those relationships.

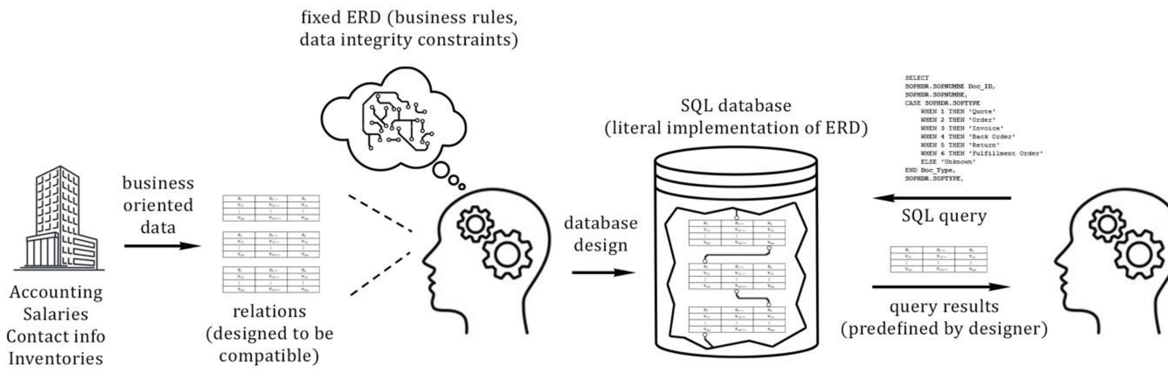
While the relation in AQL is equivalent to Codd's, our definition of the relation is based on the model of De Hann [10] whose defined a modified form of a relation and called it a *skeleton*. More importantly, our conception of the relational database model is significantly different as we shall see.

Codd's Database Model

Codd defined a *database* as a set of relations and their logical connections under the control of a data sublanguage (such as SQL) [1][4]. The relations in his model were time-varying, but their number, form, and interrelationships were fixed. In Codd's model, a database could be viewed as a fixed circuit in which relations are conduits for data, and the DBMS controls the data flow [3]. An entity-relationship diagram, or *ERD*, is used to model a relational database. ERDs depict a kind of data circuit diagram and serve as a description of the components of the database and their interrelationships (see figure 3).

However, Codd's model was not only about abstract mathematical structures and mathematical models for query languages. It was also about controlling the user/machine interface. To this end, he added data integrity constraints to his model. These constraints defined the access privileges of users, tight rules governing the operations performed on the data by users, and the rules for how users define every data structure in the database. The relational model regulated the user's relationship with the database. In other words, the relational model was about social engineering.

Figure 3. Codd's relational database in diagrammatic form illustrates the process of database design and access.


fixed ERD (business rules, data integrity constraints)
business oriented data
Accounting
Salaries
Contact info
Inventories
relations (designed to be compatible)
database design
SQL database (literal implementation of ERD)
SQL query
query results (predefined by designer)

```
SELECT
SOPHDR.SOPNUMBE Doc_ID,
SOPHDR.SOPNUMBE,
CASE SOPHDR.SOPTYPE
    WHEN 1 THEN 'Quote'
    WHEN 2 THEN 'Order'
    WHEN 3 THEN 'Invoice'
    WHEN 4 THEN 'Back Order'
    WHEN 5 THEN 'Return'
    WHEN 6 THEN 'Fulfillment Order'
    ELSE 'Unknown'
END Doc_Type,
SOPHDR.SOPTYPE,
```

Moreover, Codd's social engineering criteria guaranteed that critical data of any kind, be it financial or organizational, is rigorously protected while still permitting appropriate access. Five forms of data integrity constraints were developed to protect data from malicious users as well as control the access of authorized users [4]. The relational operations were also governed by the integrity constraints so that these operations would produce consistent results while not damaging existing data.

In addition, most users were presumed to be either averse or uninterested in understanding the underlying mathematical concepts of relational systems, and hence needed a layer of control, supervision, and a query tool they would be willing to understand. The data integrity constraints were embedded in the data sublanguage to make it extremely difficult to avoid them [3]. Thus, the integrity constraints also adhered to the notion of data independence. The database structure used to store the data, and the data sublanguage used to access and protect the data were self-contained and independent of any host language within which they were accessed.

We should acknowledge that the notion of a fixed ERD implemented in SQL has loosened substantially in recent years. Today, the SQL user can add tables more easily than in earlier versions. Also, the user can perform operations on those tables without restriction. However, the issue of typing new tables persists in that the user must supply typing information for new tables inline with the SQL commands that create them. This issue has, in most cases, been dealt with by AQL's automatic type system discussed below.

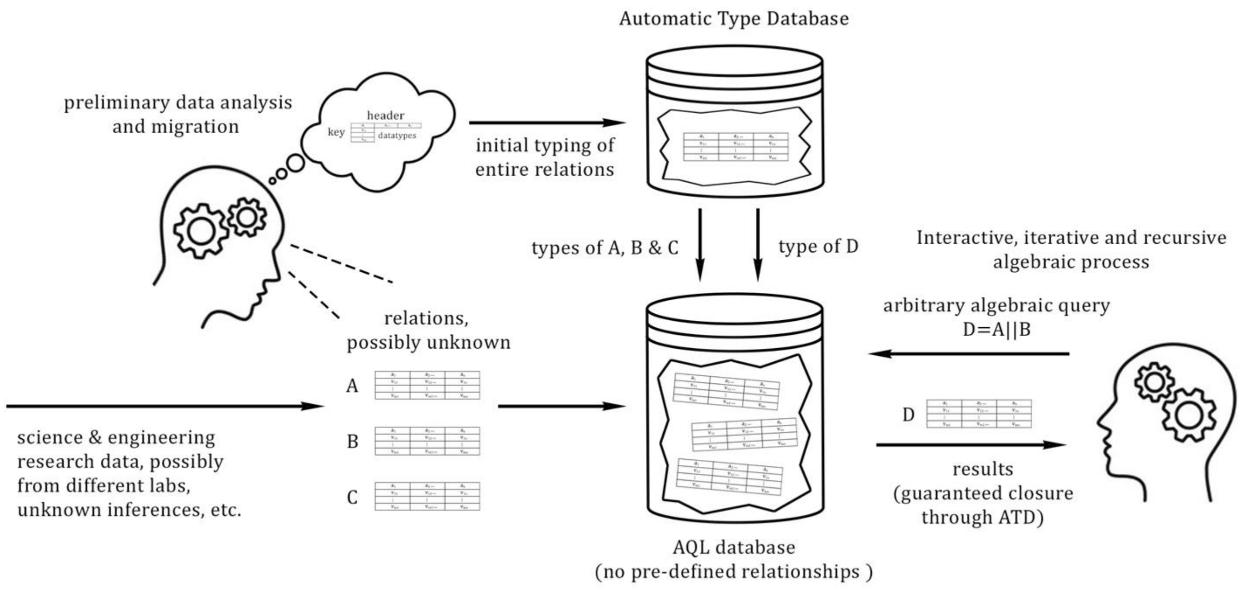
AQL's Database Model

Fundamental differences with Codd's relational model have shaped the AQL database model. The initial purpose of AQL was to support scientific research, not business systems. Thus, we re-examined Codd's social engineering concepts of data integrity by stressing free access to data and we make different assumptions about the users. To do this, we had to make the database model much more malleable, moving from a fixed system to a more dynamic concept.

In an AQL database the number of relations, their interrelationships, inferences, and data can vary with time. The number of attributes of a relation can grow or decline, change, and shift values. The relation can change its type and get a new primary key. Referential relationships can be broken and new one established. The relation becomes a true variable.

Thus, we have dispensed with a fixed ERD and relationships, and instead, given each relation a datatype used to compute the inferences needed to perform queries. *We type entire relations as individual units in the database*. While this seems trivial, it permits the flexibility that we wanted in our database model. Whenever an AQL operation creates a new relation, its datatype is also created and stored in an Automatic Type Database (ATD) for later use. This process is essential to AQL's ability to perform algebraic operations upon relations (see figure 4).

Figure 4. The AQL Database model in diagrammatic form illustrates the process of database design and access.


Automatic Type Database
preliminary data analysis and migration
header
key
datatypes
initial typing of entire relations
types of A, B & C
type of D
Interactive, iterative and recursive algebraic process
arbitrary algebraic query D=A||B
relations, possibly unknown
A
B
C
D
science & engineering research data, possibly from different labs, unknown inferences, etc.
results (guaranteed closure through ATD)
AQL database (no pre-defined relationships)

In our model, referential connections between relations may not initially be known in an AQL database. Instead, they can be tested by means of the AQL operators. AQL's algebraic operators use the datatypes of the relations to determine potential referential connections between them. In other words, the suite of AQL operators is used to *mine* the database to *verify and quantify* the interrelationships between or within relations. In addition, the AQL assignment operators can be used to save intermediate and final results of a query at almost any point in the search expression. These intermediate results can be used to locate and diagnose errors. This constitutes what we call the *open architecture* and *open workflow* of the ADBMS.

Furthermore, we implement relational algebra in place of predicate calculus, thus branching from the early path taken in the development of SQL. AQL supports the basic operations for search that Codd specified. However, it emphasizes computation over description and therefore implements relational algebra, which is preferable for a computational approach in a data sublanguage. AQL has a full set of relational operators and many non-relational ones. As we will see, AQL's algebraic syntax allows querying concurrently with numerical computation, which Codd specifically rejected [3].

Thus, the AQL relational model grew from Codd's, but stresses a more intuitive workflow, simplicity, computation, speedy access, and encourages experimentation with data. It gives the user full access to the data and assumes the user can understand the mathematical details of query operations. Yet, its syntax is simple enough so that users don't necessarily have to understand all the details.

3.3 Data Sublanguages and Natural Language

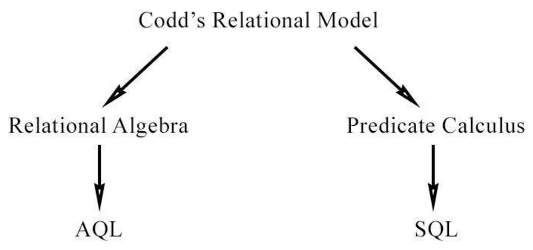


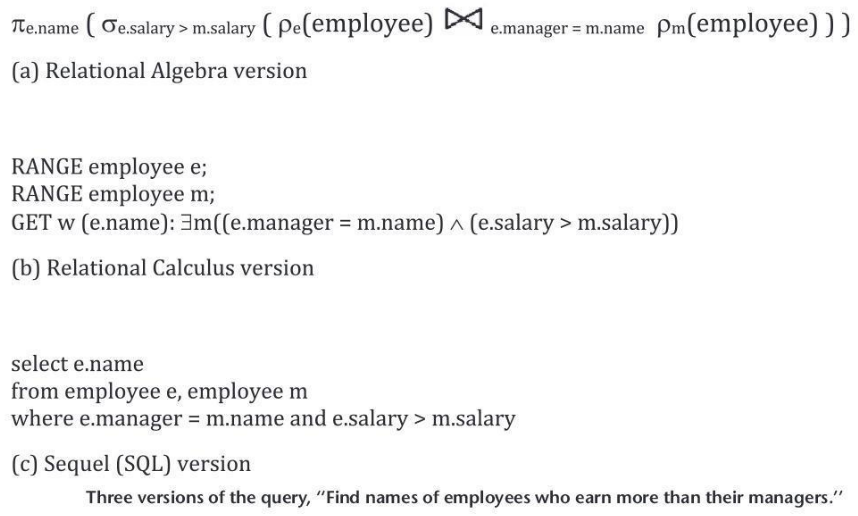
Figure 5. Codd Model, Sublanguages and Resultant Query Languages. Here we see the point of departure for AQL within his model.

Codd created models for *data sublanguages* [5] that are used to query databases. He showed how to determine the degree to which they adhere to his relational paradigm. He created two of these models: one called relational algebra, and the other called *Alpha* that was based on predicate calculus. Structured Query Language (SQL) was modeled after Alpha [6]. AQL, written by the author of this paper, was modeled after relational algebra.

SQL was designed to implement Codd's relational model, and today it is the *lingua franca* of the relational database world. It was originally pioneered by Donald Chamberlin and Ray Boyce while working under Codd at the IBM research facility in San Jose, California in 1972 [6]. SQL itself was derived from SEQUEL: Structured English Query Language [8]. Early efforts in SEQUEL development reflected their choices of English language programming.

With SQL, Chamberlain and Boyce hoped to put database query languages into the hands of a broad audience [6]. Chamberlain studied three candidate syntaxes shown in Figure 6 that would query a database consisting of the single relation shown in Figure 2. Each example would answer the following query: "Which employees make more than their managers?" He was trying to show here that while relational algebra and predicate calculus were concise, they were steeped in technically complicated mathematical notation. In SQL, they believed they had created a language free of burdensome mathematical notation and concepts, while still conforming to Codd's relational model. They hoped [6]…

> "…that, with a little practice, users could learn to read queries like this almost as though they were English prose. Their example shown in [*figure 4*] could be read as follows: 'Find an employee (let's call him '*e*') and another employee (let's call him '*m*') where *e*'s manager matches *m*'s name (in other words, *e*'s manager is *m*) and *e*'s salary is greater than *m*'s salary (in other words, *e* earns more than his manager); then print *e*'s name (for every such employee)."



$\pi_{e.name}$ ($\sigma_{e.salary > m.salary}$ (ρ_e(employee) $\bowtie_{e.manager = m.name}$ ρ_m(employee)))

(a) Relational Algebra version

```
RANGE employee e;
RANGE employee m;
GET w (e.name): ∃m((e.manager = m.name) ∧ (e.salary > m.salary))
```

(b) Relational Calculus version

```
select e.name
from employee e, employee m
where e.manager = m.name and e.salary > m.salary
```

(c) Sequel (SQL) version

Three versions of the query, ''Find names of employees who earn more than their managers.''

Figure 6. Chamberlain's "Three Versions of the Query"

To further enforce their concept of social engineering, Codd *et al.* also felt that a query language should describe the information sought but not provide a detailed plan for how to find that information. They saw the interpreter's job as translating the declarative query into an optimized search, thus hiding procedural details from the user [3]. Further, Codd intentionally rejected computation in his relational model, specifically in his models of data independence and integrity [4].

Thus, SQL was designed around the notion that a query language based on a natural language would make queries easier for humans to read and write than either relational algebraic or predicate calculus queries. In addition, it would exclude mathematical details irrelevant to making a query. SQL is as close a translation of predicate calculus into an English-like language as possible, creating a query language with English-like syntax.

As SQL evolved, it conformed more closely to Codd’s relational model via integrity constraints and left behind the trappings of predicate calculus becoming a more English-like natural language [5].

4 AQL: An Implementation of Relational Algebra

We maintain that the English syntax of SQL, is more awkward and confusing than algebraic expressions because they encumber mathematical expressions with extraneous semantics and syntax that require additional mental gymnastics to frame. Moreover, using an English syntax strips the mathematical notation of its natural and elegant concision. By eliminating the mathematical syntax, we believe that Codd et al. removed mathematics' useful and intuitive qualities from query languages.

In contrast to SQL, AQL's syntax has a small set of concise and intuitive rules that can be easily used to create query expressions. Using these rules as implemented in AQL, we can generate an infinite number of query expressions. This is because the rules mutually feed into one another permitting the indefinite expansion of AQL queries. This is known in linguistics as recursion [11]. For example, consider two of the rules taken from the formal definition of the AQL language that follows immediately after this section.

1) *If L is an AQL fundamental operator and Q an AQL search expression, then $L = Q$ is an AQL search expression*
2) *If Q and P are AQL search expressions, and $+$ is a binary AQL operator, then $Q + P$ is also an AQL search expression.*

Rule 1 means that any individual operand (R) can be replaced with $L= Q$ in any AQL expression. likewise, rule 2 means that any individual operand (R) can be replace with $Q+P$. The following expansion of a simple query illustrates the process of one rule feeding another, generating longer and more complex expressions:

$L=Q$	*Rule 1*
$L=R+P$	*Rule 2* $Q \rightarrow R+P$
$L=(R+P)+W$	*Rule 2* $R+P \rightarrow (R+P)+W$
$L=(M=R+P)+W$	*Rule 1* $R+P \rightarrow M=R+P$
$L=(M=R+P)+(X+Y)$	*Rule 2* $W \rightarrow X+Y$
$L=(M=R+P)+(Z=X+Y)$	*Rule 1* $X+Y \rightarrow Z=X+Y$
and so on…	

In addition, it is possible to refactor a complex AQL expression into a sequence of simpler expressions that still gives the same result:

L=(M=R+P)+(Z=X+Y)	*Single AQL expression*
M=R+P	*Refactoring from the*
Z=X+Y	*Inside out.*
L=M+Z	*The result is identical*

Of course, both English and SQL have recursion. Recursion is a property of all natural languages and algebra. SQL's recursion might be more limited than in English. But the extent of SQL's recursion is of no importance because the real problem lies in the amount of unneeded baggage that is introduce each time a recursion is performed. For example, imagine the complexity resulting from putting any of the command shown in figure 16 into any of the other commands. We contend that using English language forms for what are essentially mathematical objects creates a burdensome query language.

Chamberlain's example above is used to imply that relational algebra is incomprehensible compared to SEQUEL or SQL. We took a different approach to implement relational algebra, which resulted in an even simpler form suitable for command-line input. To this end, note that the relational algebraic expression found in Figure 6:

$$\pi_{e.name}\left(\sigma_{e.salary>m.salary}\left(\rho_e(employee) \bowtie_{e.manager=m.name} \rho_m(employee)\right)\right)$$

can be re-stated with a little massaging into AQL as:

$$\left((Employee \;\&\&\; Manager) :: @esalary > \; @msalary\right) :> @name$$

In the AQL expression above, && is a binary operator that combines the two relations Employee and Manager, **::** and **:>** are scalar operators, and Employee and Manager are references to AQL tables. Finally, @esalary, @msalary, and @name are assumed to be attributes of relations Employee and Manager. This operation is more fully explained and computed subsequently.

In our social engineering model, AQL's users do not benefit from a built-in lack of computation or hiding mathematical and operational details. They are treated as intelligent practitioners who want to use the power of mathematics, with all the benefits and caveats that follow from the openness of AQL's design.

5 *Formal AQL Definition*

Now, we move on from our introduction to a formal definition of AQL. AQL is a query language based on relational algebra, supporting the entire set of relational operators, including additional non-relational operators. It uses Leibnitz notation for function operators, standard notation for binary operators, and scalar operators as motivated by those from linear spaces. The following definition is recursive, and the depth of embedding is limited only by the machine resources available to the user.

The following is an abstract language definition. To simplify this definition, we will group the binary and unary operators into the following categories, and introduce a simplified meta-language or pseudo-code:

$+$ representing all join operators

$$\&\& \quad || \quad <\&\& \quad \&\&> \quad <|| \quad ||> \quad ** \quad @\&\& \quad @|| \quad >>$$

$\rightarrow$ representing all scalar operators

$$:: \quad :> \quad >< \quad -> \quad <>$$

$=$ representing all assignment operators

$$= \quad \hat{}= \quad \&\&= \quad ||=$$

Also note the following conventions for search expressions:

- L for the name of an AQL file, containing a single relation on disk, which is a basic AQL search expression and is also called the *fundamental operator*.
- $\$L$ for an AQL symbol, a single relation in memory, also a basic AQL search expression or fundamental operator.
- Q and P for AQL search expressions, which can be as simple as a fundamental operator or a complex expression with many operators and operands.
- $F(\bullet)$ for function operators, which are AQL search expressions

Here are the AQL pseudo code conventions for components of the language used to build search expressions:

- S for scalar expressions (including strings, mathematical and statistical expressions)
- C for a conditional expression
- CS for a conditional statement
- A for an attribute list (@a:@b:@c for example)
- AL for an attribute expression list

Here are the rules that define AQL as an algebraic language over the set of relations under the following operations:

1. L and *$L* are AQL search expressions
2. L = Q is an AQL search expression
3. $Q + P$ is an AQL search expression
4. $Q \rightarrow C,\ Q \rightarrow A\ \ or\ \ Q \rightarrow AL$ are AQL search expressions
5. *F(Q,…)* and *F(Q, P,…)* are AQL search expressions. Their comma-separated parameters *(…)* may include one or more scalar expressions *S,* attribute expression lists *AL*, conditional expressions *C*, conditional statements *CS*, and AQL file names *L*
6. (*Q)* and *[Q]* are AQL search expressions. Thus, AQL permits the use of parentheses and brackets in expressions to override the rules of precedence for AQL operators

(AQL is built upon the BerkeleyDB key-value store and is not innovative at that level).

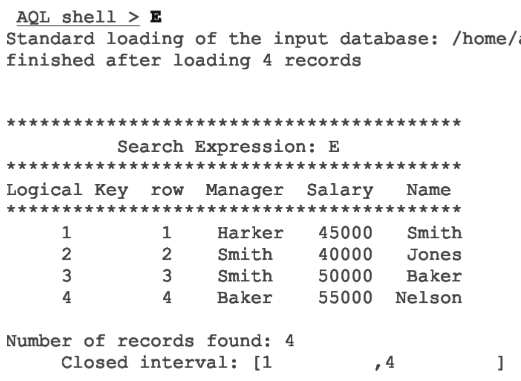
5.1 AQL Usage

While the basic concepts of AQL usage are straightforward, an in-depth discussion is outside the scope of this paper, but we will now briefly expand on this language definition with three straightforward use cases that demonstrate the rules. *L*, *Q*, and *P* are the same abstract symbols used in the formal definition above in the examples below. They are realized in specific examples in the names E and F. We repeat the rules demonstrated for emphasis. The rules are repeated for convenience.

1. *L is an AQL search expression*. The relation is the fundamental unit of an AQL expression. Therefore, consider the entry point to be *L* (the name of an AQL file storing a relation also called the *fundamental operator*). For example, the following code snippet uses only the name of a relation E:

```
AQL shell> E<return>
```

Submitting it to the AQL interpreter will result in its contents being loaded into memory by the ADBMS and displayed on the screen. (Note that this operation does not change the file contents in any way or rewrite it to disk.)



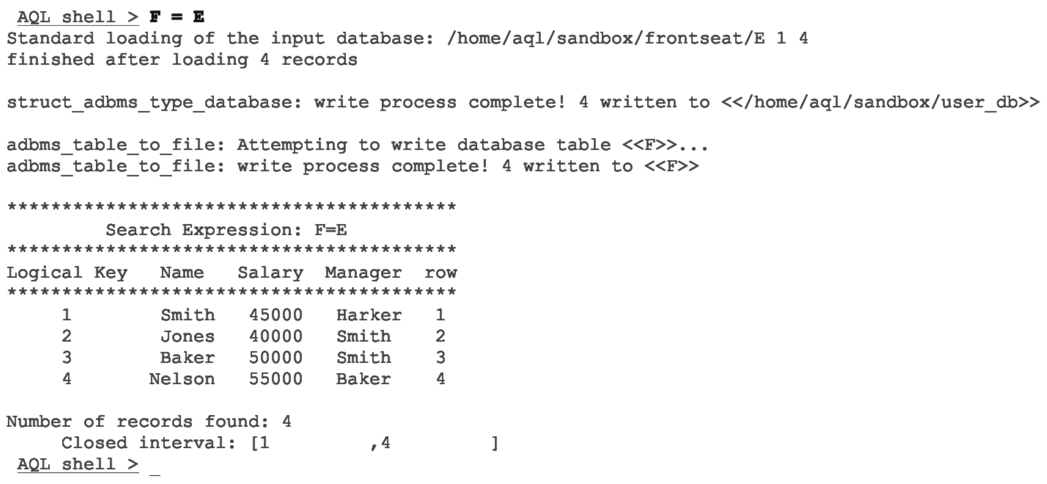
```
 AQL shell > E
Standard loading of the input database: /home/aql/sandbox/frontseat/E 1 4
finished after loading 4 records


****************************************
          Search Expression: E
****************************************
Logical Key  row  Manager  Salary   Name
****************************************
     1        1    Harker   45000   Smith
     2        2    Smith    40000   Jones
     3        3    Smith    50000   Baker
     4        4    Baker    55000  Nelson

Number of records found: 4
     Closed interval: [1        ,4        ]
```

Figure 7. The Fundamental Search Expression

This query was performed in the AQL shell. On the table's left side, you will notice a column with the name "Logical Key." In AQL, the term logical key is equivalent to the standard term "primary key." This column is added to the table as a convenience. The logical key often consists of more than one attribute. In this example however, "row" is the only attribute in the logical key. The input tables are not changed by this operation. It still contains only four attributes. The closed interval is the range of the logical key values.

2. *L = Q is an AQL search expression.* The *assignment operator*, **=**, is one of the most essential operators found in AQL. It is a binary operator that allows the user to store the results of an AQL search expression either to disk or memory. In the following simplest example, we will make a copy of E on the disk and name it F. We will also create the symbol $F for E.

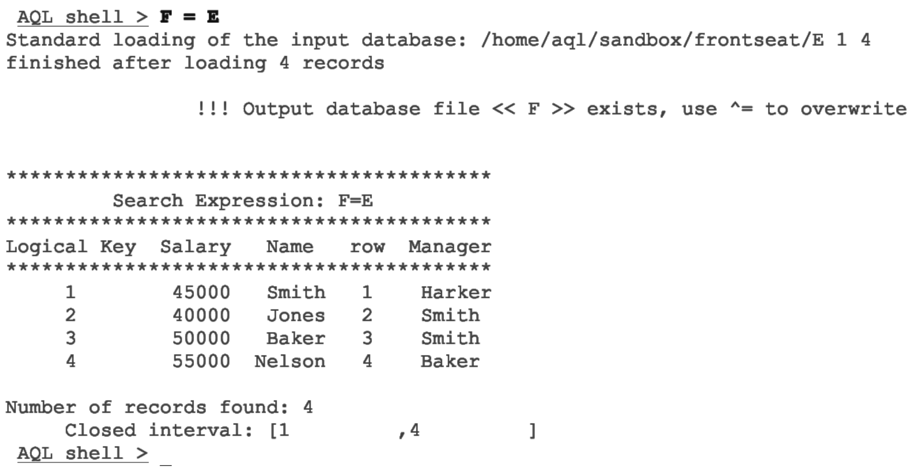
```
AQL shell> F = E<return>
```



```
 AQL shell > F = E
Standard loading of the input database: /home/aql/sandbox/frontseat/E 1 4
finished after loading 4 records

struct_adbms_type_database: write process complete! 4 written to <</home/aql/sandbox/user_db>>

adbms_table_to_file: Attempting to write database table <<F>>...
adbms_table_to_file: write process complete! 4 written to <<F>>

****************************************
         Search Expression: F=E
****************************************
Logical Key   Name   Salary  Manager  row
****************************************
     1        Smith   45000   Harker   1
     2        Jones   40000   Smith    2
     3        Baker   50000   Smith    3
     4       Nelson   55000   Baker    4

Number of records found: 4
     Closed interval: [1         ,4        ]
 AQL shell > _
```

Figure 8. The Assignment Operator

There are safeguards against a file overwrite if we try to write to E again:



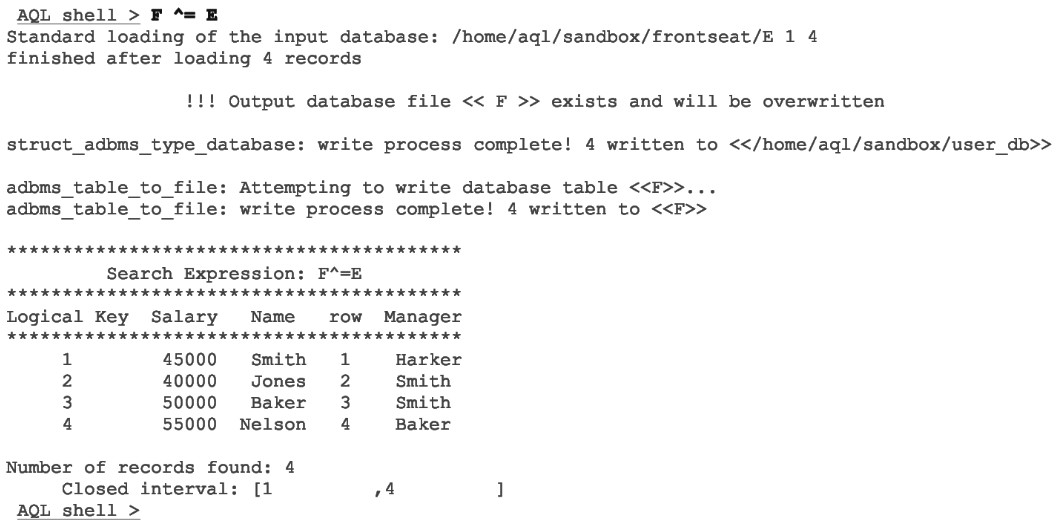
```
 AQL shell > F = E
Standard loading of the input database: /home/aql/sandbox/frontseat/E 1 4
finished after loading 4 records

                !!! Output database file << F >> exists, use ^= to overwrite


*****************************************
         Search Expression: F=E
*****************************************
Logical Key  Salary   Name   row  Manager
*****************************************
     1        45000   Smith   1    Harker
     2        40000   Jones   2    Smith
     3        50000   Baker   3    Smith
     4        55000  Nelson   4    Baker

Number of records found: 4
     Closed interval: [1         ,4         ]
 AQL shell > _
```

Figure 9. Unintentional Overwrite Protection

Note that the interpreter will always try to return the result of a search even if the expression produced an error. In this case, overwrite can be forced with the *overwrite assignment* operator ^= thus:

```
AQL shell> F ^= E<return>
```

With the results:

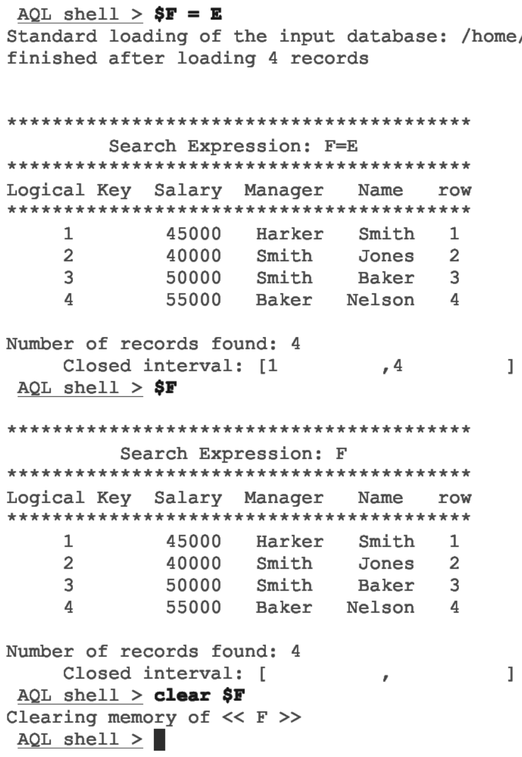


```
 AQL shell > F ^= E
Standard loading of the input database: /home/aql/sandbox/frontseat/E 1 4
finished after loading 4 records

                !!! Output database file << F >> exists and will be overwritten

struct_adbms_type_database: write process complete! 4 written to <</home/aql/sandbox/user_db>>

adbms_table_to_file: Attempting to write database table <<F>>...
adbms_table_to_file: write process complete! 4 written to <<F>>

*****************************************
         Search Expression: F^=E
*****************************************
Logical Key  Salary   Name   row  Manager
*****************************************
     1        45000   Smith   1    Harker
     2        40000   Jones   2    Smith
     3        50000   Baker   3    Smith
     4        55000  Nelson   4    Baker

Number of records found: 4
     Closed interval: [1         ,4         ]
 AQL shell >
```

Figure 10. The Overwrite Assignment Operator

We can use the assignment operator to create AQL symbols that reference relations that have been stored in memory. It is used in precisely the same way we created the disk file F, i.e. $F = E. Also shown is how the symbol can be referenced as a simple AQL expression, and how it can be cleared from memory:

```
AQL shell> $F = E<return>
```



```
 AQL shell > $F = E
Standard loading of the input database: /home/aql/sandbox/frontseat/E 1 4
finished after loading 4 records


*****************************************
         Search Expression: F=E
*****************************************
Logical Key  Salary  Manager   Name   row
*****************************************
     1        45000   Harker   Smith   1
     2        40000   Smith    Jones   2
     3        50000   Smith    Baker   3
     4        55000   Baker   Nelson   4

Number of records found: 4
     Closed interval: [1         ,4         ]
 AQL shell > $F

*****************************************
          Search Expression: F
*****************************************
Logical Key  Salary  Manager   Name   row
*****************************************
     1        45000   Harker   Smith   1
     2        40000   Smith    Jones   2
     3        50000   Smith    Baker   3
     4        55000   Baker   Nelson   4

Number of records found: 4
     Closed interval: [          ,          ]
 AQL shell > clear $F
Clearing memory of << F >>
 AQL shell > 
```

Figure 11. Using Symbols: Assignment, Access, and Removal from the Symbol Table

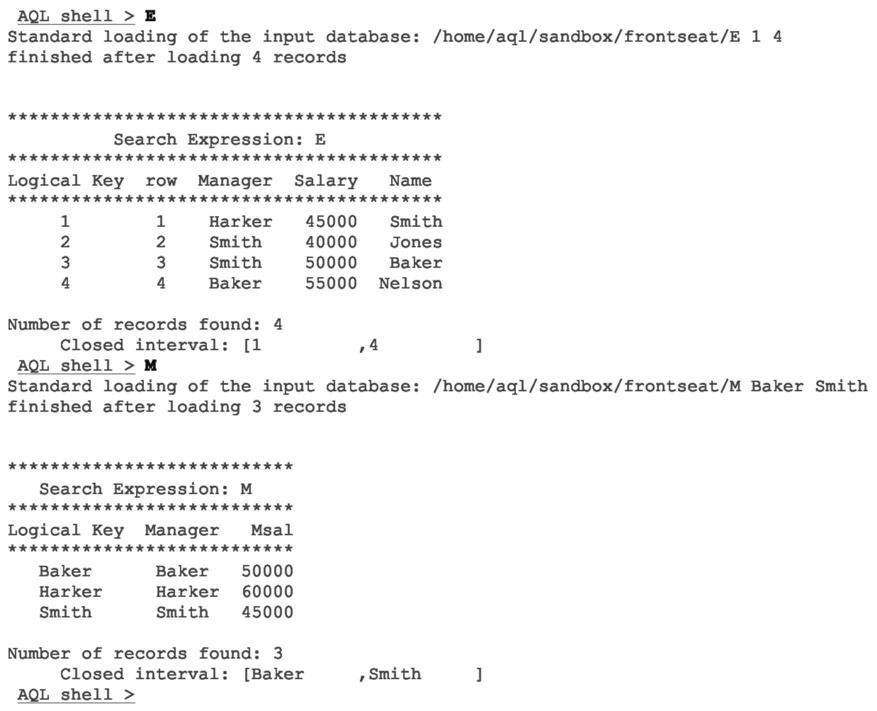
The AQL shell is equipped with many of the standard Unix shell commands (**ls**, **cd**, **rm**, etc.).

3. *Q + P is an AQL search expression.* For example, the *inner join* && operator matches common attributes between each record in E and M. Here we view the contents of E and M by stating them as fundamental operators:

```
AQL shell> E<return>
```

and

```
AQL shell> M<return>
```

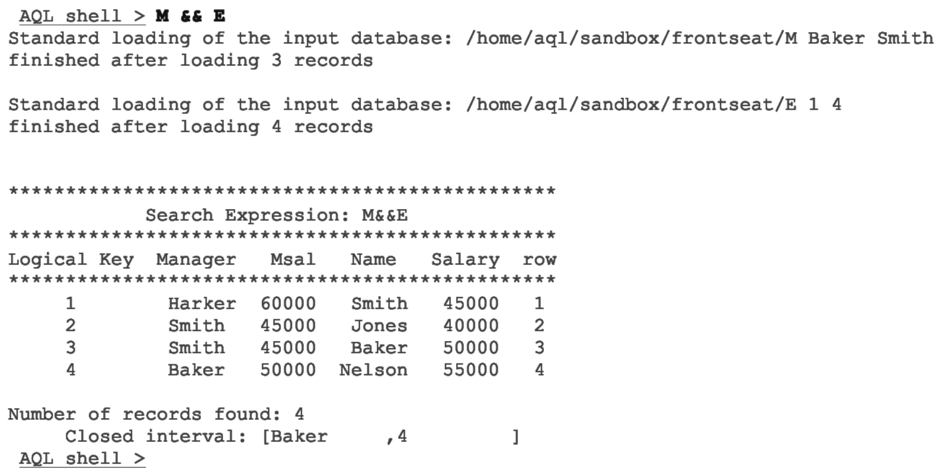


```
 AQL shell > E
Standard loading of the input database: /home/aql/sandbox/frontseat/E 1 4
finished after loading 4 records


*****************************************
          Search Expression: E
*****************************************
Logical Key  row  Manager  Salary   Name
*****************************************
     1        1    Harker   45000   Smith
     2        2    Smith    40000   Jones
     3        3    Smith    50000   Baker
     4        4    Baker    55000  Nelson

Number of records found: 4
     Closed interval: [1         ,4         ]
 AQL shell > M
Standard loading of the input database: /home/aql/sandbox/frontseat/M Baker Smith
finished after loading 3 records


***************************
   Search Expression: M
***************************
Logical Key  Manager   Msal
***************************
   Baker      Baker   50000
   Harker     Harker  60000
   Smith      Smith   45000

Number of records found: 3
     Closed interval: [Baker     ,Smith     ]
 AQL shell >
```

Figure 12. Displaying E and M

Note that E and M share the "Manager" attribute in common so that the join will likely produce results:

```
AQL shell> M && E<return>
```



```
 AQL shell > M && E
Standard loading of the input database: /home/aql/sandbox/frontseat/M Baker Smith
finished after loading 3 records

Standard loading of the input database: /home/aql/sandbox/frontseat/E 1 4
finished after loading 4 records


***********************************************
            Search Expression: M&&E
***********************************************
Logical Key  Manager   Msal   Name   Salary  row
***********************************************
     1        Harker  60000   Smith   45000   1
     2        Smith   45000   Jones   40000   2
     3        Smith   45000   Baker   50000   3
     4        Baker   50000  Nelson   55000   4

Number of records found: 4
     Closed interval: [Baker     ,4         ]
 AQL shell >
```

Figure 13. Joining E and M

Records that match their common attributes are merged and put into the output relation. In other words, this operation will match and merge records that have the same value for the "Manager" attribute. These records correspond to Codd's inferences made from the input facts in M and E. The above command, again, does not change either E or M or create a new relation, and in SQL parlance is simply a view of what that join would be. If we wish to save these results, we use an assignment operator to create a new relation storing the results in our database. The following command would accomplish that task:

```
AQL shell>A = M && E<return>
```

The AQL file A would store the output of `M && E`.

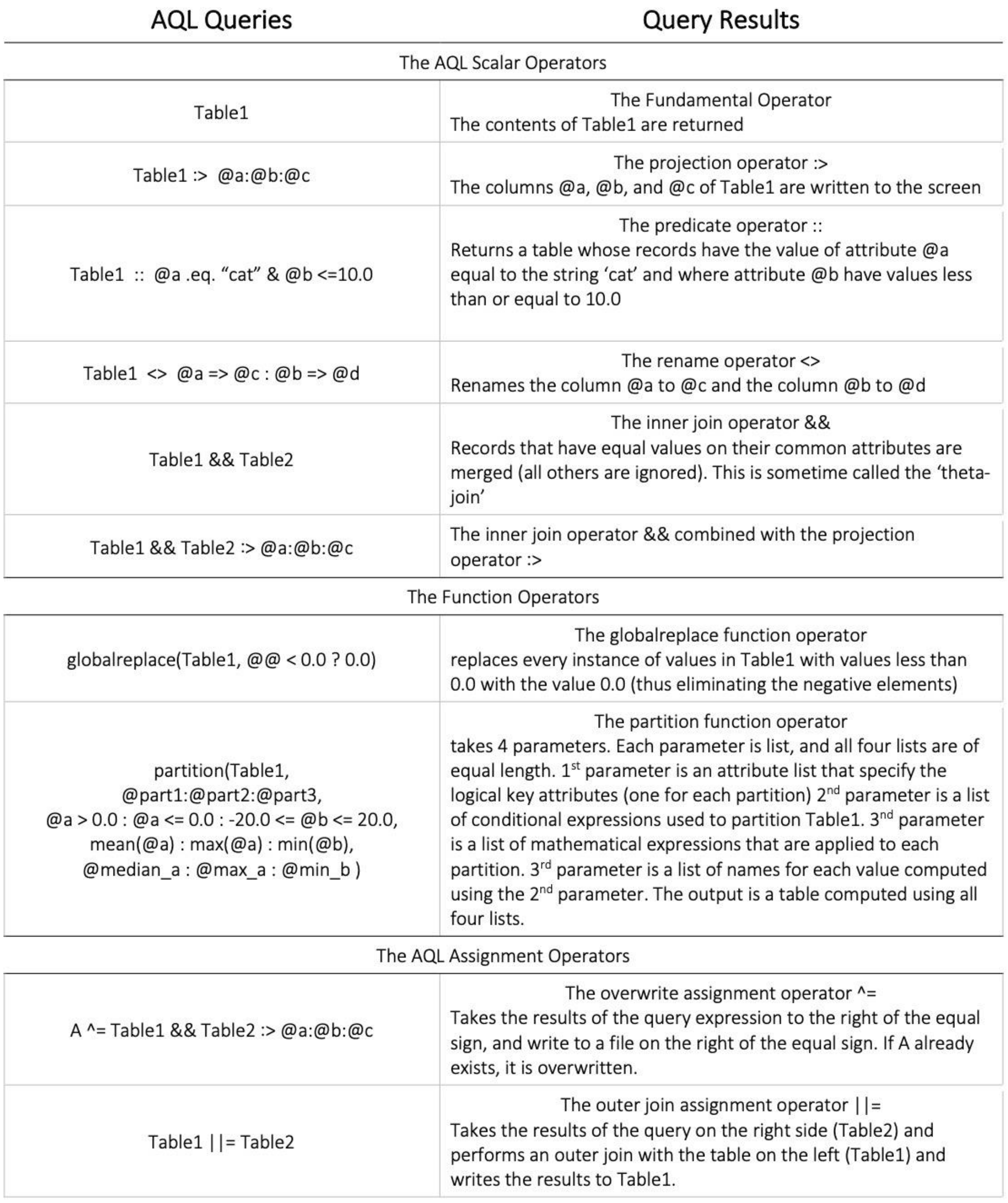
5.2 *Comparing AQL's with SQL's syntax*

Now we will contrast the syntax of AQL with that of SQL. Figure 15 shows a table of AQL expressions, and their query results. Figure 16 is the translation of each of the expressions in Table I into an equivalent SQL statement. The SQL translations are true to the implementation of SQL by Oracle.

The details of implementing the database required to render each of the expressions as operational queries is not provided. To assist the reader unfamiliar with the details of AQL syntax, here is a summary of language objects that appear in the AQL queries below:

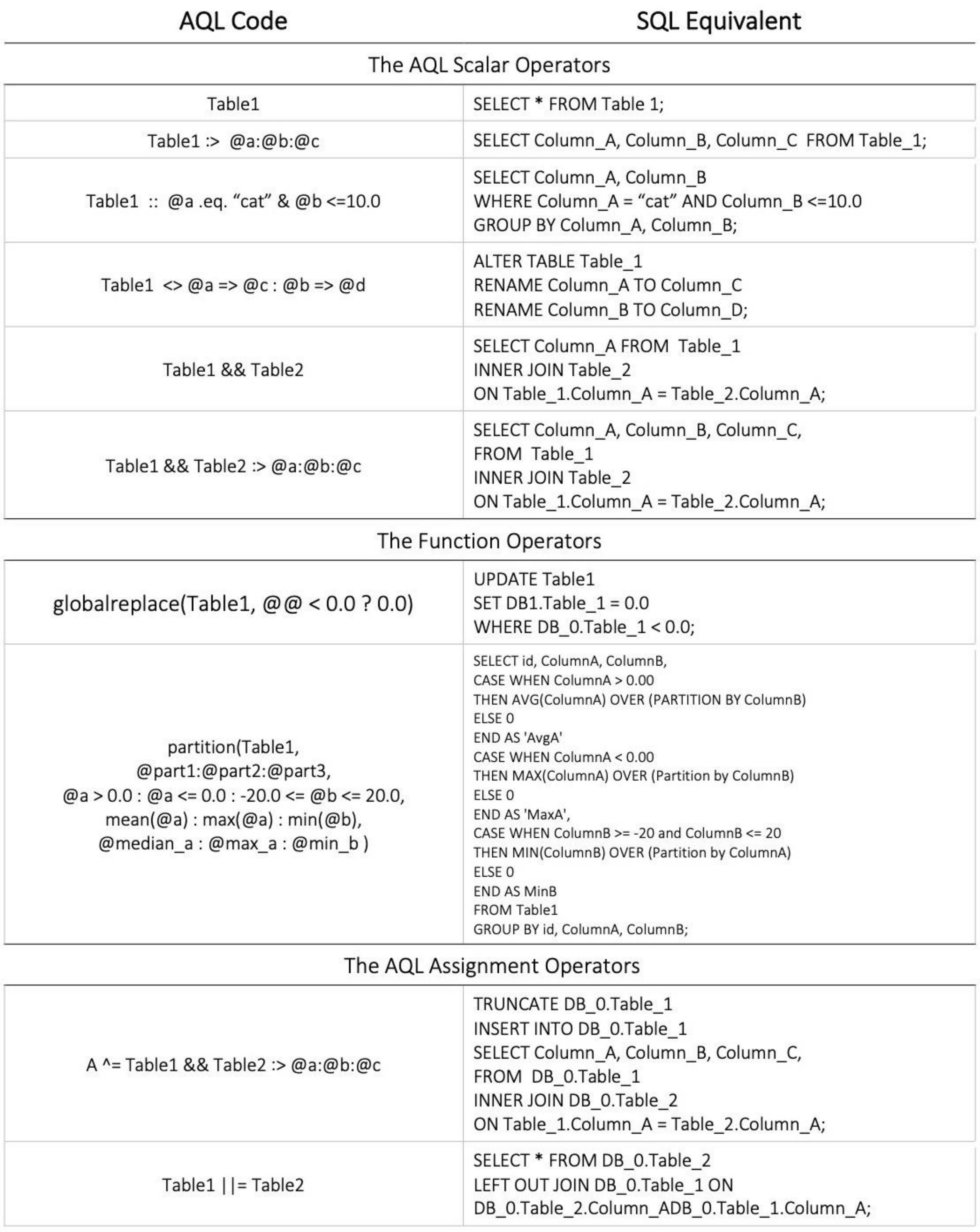
- @a, @b, @c are the attributes of a table (relation). In AQL, we think in terms of attribute names when referring to the columns of a table. This is obvious as the notion of a column originates with the table which is only a visualization of a relation. However, relations don't have columns, they have domains, and those domains are given names called attributes. In AQL we think in terms of attributes. So, its syntax combines the ampersand and an attribute name to refer to what in SQL would be called a column.
- .eq. and <= are the ordering operators that test the relationship between two quantities (strings for the former, and numbers for the latter). AQL has a full suite of ordering operators for strings and numbers.
- AQL uses commas and colons as delimiters in lists of various kinds. So @a:@b:@c is a list of attributes (suitable for the projection operator). Commas are used as separators for the parameters of the function operators.
- The scalar and binary operators act on relations and are given two or three character symbols: &&, :>,::, ||=

Figure 15. A table of AQL queries and their meaning is defined by their query results. These are translated into SQL in figure 16.



AQL Queries	Query Results
The AQL Scalar Operators	
Table1	The Fundamental Operator The contents of Table1 are returned
Table1 :> @a:@b:@c	The projection operator :> The columns @a, @b, and @c of Table1 are written to the screen
Table1 :: @a .eq. "cat" & @b <=10.0	The predicate operator :: Returns a table whose records have the value of attribute @a equal to the string 'cat' and where attribute @b have values less than or equal to 10.0
Table1 <> @a => @c : @b => @d	The rename operator <> Renames the column @a to @c and the column @b to @d
Table1 && Table2	The inner join operator && Records that have equal values on their common attributes are merged (all others are ignored). This is sometime called the 'theta-join'
Table1 && Table2 :> @a:@b:@c	The inner join operator && combined with the projection operator :>
The Function Operators	
globalreplace(Table1, @@ < 0.0 ? 0.0)	The globalreplace function operator replaces every instance of values in Table1 with values less than 0.0 with the value 0.0 (thus eliminating the negative elements)
partition(Table1, @part1:@part2:@part3, @a > 0.0 : @a <= 0.0 : -20.0 <= @b <= 20.0, mean(@a) : max(@a) : min(@b), @median_a : @max_a : @min_b)	The partition function operator takes 4 parameters. Each parameter is list, and all four lists are of equal length. 1st parameter is an attribute list that specify the logical key attributes (one for each partition) 2nd parameter is a list of conditional expressions used to partition Table1. 3nd parameter is a list of mathematical expressions that are applied to each partition. 3rd parameter is a list of names for each value computed using the 2nd parameter. The output is a table computed using all four lists.
The AQL Assignment Operators	
A ^= Table1 && Table2 :> @a:@b:@c	The overwrite assignment operator ^= Takes the results of the query expression to the right of the equal sign, and write to a file on the right of the equal sign. If A already exists, it is overwritten.
Table1 ||= Table2	The outer join assignment operator ||= Takes the results of the query on the right side (Table2) and performs an outer join with the table on the left (Table1) and writes the results to Table1.

Figure 16. The translation from AQL to their SQL equivalent. In some cases, the translation may be approximate.



AQL Code	SQL Equivalent
The AQL Scalar Operators	
Table1	SELECT * FROM Table 1;
Table1 :> @a:@b:@c	SELECT Column_A, Column_B, Column_C FROM Table_1;

Table1 :: @a .eq. “cat” & @b <=10.0	SELECT Column_A, Column_B
	WHERE Column_A = “cat” AND Column_B <=10.0
	GROUP BY Column_A, Column_B;

Table1 <> @a => @c : @b => @d	ALTER TABLE Table_1 RENAME Column_A TO Column_C RENAME Column_B TO Column_D;

Table1 && Table2	SELECT Column_A FROM Table_1
	INNER JOIN Table_2
	ON Table_1.Column_A = Table_2.Column_A;

Table1 && Table2 :> @a:@b:@c	SELECT Column_A, Column_B, Column_C,
	FROM Table_1
	INNER JOIN Table_2
	ON Table_1.Column_A = Table_2.Column_A;

The Function Operators	

globalreplace(Table1, @@ < 0.0 ? 0.0)	UPDATE Table1
	SET DB1.Table_1 = 0.0
	WHERE DB_0.Table_1 < 0.0;

partition(Table1,
@part1:@part2:@part3,
@a > 0.0 : @a <= 0.0 : -20.0 <= @b <= 20.0,
mean(@a) : max(@a) : min(@b),
@median_a : @max_a : @min_b)
	SELECT id, ColumnA, ColumnB,
	CASE WHEN ColumnA > 0.00
	THEN AVG(ColumnA) OVER (PARTITION BY ColumnB)
	ELSE 0
	END AS 'AvgA'
	CASE WHEN ColumnA < 0.00
	THEN MAX(ColumnA) OVER (Partition by ColumnB)
	ELSE 0
	END AS 'MaxA',
	CASE WHEN ColumnB >= -20 and ColumnB <= 20
	THEN MIN(ColumnB) OVER (Partition by ColumnA)
	ELSE 0
	END AS MinB
	FROM Table1
	GROUP BY id, ColumnA, ColumnB;

The AQL Assignment Operators	

A ^= Table1 && Table2 :> @a:@b:@c	TRUNCATE DB_0.Table_1
	INSERT INTO DB_0.Table_1
	SELECT Column_A, Column_B, Column_C,
	FROM DB_0.Table_1
	INNER JOIN DB_0.Table_2
	ON Table_1.Column_A = Table_2.Column_A;

Table1 ||= Table2	SELECT * FROM DB_0.Table_2
	LEFT OUT JOIN DB_0.Table_1 ON
	DB_0.Table_2.Column_ADB_0.Table_1.Column_A;

5.3 AQL Operators

Assignment Operators enable the user to write query results into new or pre-existing relations. The assignment operator was suggested in a textbook by Marvin Perlman [12]:

- Simple Assignment P = Q
- Overwrite Assignment P ^= Q
- Intersection Assignment P &&= Q
- Union Assignment P ||= Q

Join Operators take records from two relations**,** joining them together by merging their records:

- Union Join **P || Q**
- Intersection (Natural) Join **P && Q**
- Union Sided Join **P ||> Q** or **P <|| Q**
- Intersection Sided Join **P &&> Q** or **P <&& Q**
- Cartesian Join **P ** Q**
- Relative Complement **P -- Q**
- Distributed Join** **@&& P_1 P_2 … P_n** or **@|| P_1 P_2 … P_n**
- Template **Q >> P**

Scalar operators use conditional expressions, attribute lists, and attribute expression lists to modify records from the input relation. Their notation is a linearization of Codd's original relational algebra. Consider the following expression in relational algebra extracted from figure 6. This is the projection operator:

$$\pi_{e.name}(E)$$

The corresponding AQL expression uses AQL's projection operator :> as follows:

E :> @name

We have moved the subscript up to the same line and created a binary operation between E and @name. Using the symbol :> to represent a binary projection operator that corresponds to π in Codd's relational algebra. We call such operators, scalar operators after their counterpart (scalar multiplication) from vector spaces.

These are the *selection* scalar operators since they use ‘scalars’ that select records from the table operand Q:

- Predicate **Q :: C**
- Projection **Q :> =[]** or **Q :> ~A**
- Splice **Q >< str=<val$_1$>end=<val$_2$>**

These are the *update* scalar operators because they modify an input relation by either adding new records or changing attribute names:

- Injection **:<**
- Rename **<>**

Function Operators extend the expressiveness and utility of the AQL language. Their syntax is based on the Leibnitz standard notation for functions. In keeping with the closure requirement, the output of each operator is a relation. In most cases, the first element in the argument list (represented here by a dot •, is an AQL expression. Some of these operators take two AQL expressions as inputs.

- create(•)
- text2aql(•)
- delete(•)
- globalreplace(•)
- partition(•)
- remove(•)
- replace(•)
- quickstats(•)
- stats(•)
- update(•)
- colocate(•)
- split(•)
- convert(•)
- fuzzyjoin(•)
- extract(•)
- aql_xlsx(•)
- xlsx_aql(•)

Matrix Operators are recent additions to the AQL function operators. They either create or operate on relations of the AQL data type “matrix”.

- add(•)
- mult(•)
- *invert(•)
- diag(•)
- *solve(•)
- matrix(•)
- (* currently under development)

6 *AQL's Automatic Type Database*

Fundamentally, the above definition of AQL, and that of relational algebra, depends on the concept of closure. Closure is a property of all operators in AQL which means that no matter what kind or how many operators are involved, the outcome of any algebraic operation on one or more relations, produces a relation. Closure in AQL is achieved through defining types for relations and tracking the many forms that they take.

In programming languages, a *type-system* is a collection of rules that assign a property called *type* to language constructs such as variables, expressions, functions, or modules in a computer program. The type-system aids in the design of computer programs and it reduces errors by defining interfaces between different parts of the program and then checking that the parts have been connected in a consistent way. Thus, type systems often impose compatibility constraints between their operators and operands. For example, in Python, the expression x/y is valid so long as both x and y are floats. It fails if they are integers. The compatibility between operators and operands is also a central issue in the definition of relational operators in AQL. The Automatic Type Database (ATD) is the mechanism that checks for type compatibility in AQL.

The ATD is AQL's foremost engineering innovation (currently under a US patent held by Caltech US-2017-0043691-A1) and is critical to implementing relational algebra in AQL.

6.1 The Roots of the ATD

Implementing relational algebra in a database management system presents two difficulties for computer systems. These problems concern how to enforce algebraic closure. First, there are resource limitations associated with performing binary operations between relational operands that produce a relational result. For practical applications, these operands might be huge which places heavy demands on machine resources. Further, the results of an operation might be even larger than the inputs. Resource limitation were one of the reasons Codd originally doubted the feasibility of implementing relational algebra on the computer. Yet, resource limitations have been dealt with due to vast improvements in the manufacturing of computer memory components, and equally vast improvements in computer operating systems and memory management.

The second problem is determining the form of the result. Part of the role of typing in computer science is to assign formats to datatypes. But in the case of two arbitrary relations, *how* to track the datatype of the result of an operation must be determined, especially considering the likelihood that in general, the results are undefined. Yet whether an operator can assign a type to the result of the operation determines if that operation is closed (valid) or undefined. This is precisely the role of the ATD in AQL. Via the ATD, operators determine the types of all operands in a query expression, and compute the datatype of the result, and stores it.

6.2 How the ATD Works

AQL's built-in Automatic Typing Database (ATD) solves the problems associated with typing relations. The problems associated with typing in AQL have been solved by tracking inherent properties in a special system database set aside for this purpose. These properties are those that specify the relation that yielded Codd's data independence. In brief, relations are typed by:

- their set of domain attributes
- logical key attributes
- attribute domain constraints
- a name given to the datatype being specified

This information is packaged in records written to the ATD, which is itself a relational database in the ADBMS. The properties we used to define relational datatypes maintain data independence, since that information comes from the relation itself. Only information needed to access the data and perform query operations is used. Furthermore, as queries are processed, new types are computed concurrently with query results and stored in the ATD.

It is through the ATD that AQL enforces closure. It holds this information ready to be used by the AQL operators at the time of execution. Therefore, we think of AQL as an operator-centric language. Its operators are type sensitive and determine if the operation produces a relation that may be typed, simply empty, or left undefined. (When operations violate closure, the offending operation is said to be undefined, as with division by zero in ordinary arithmetic).

6.3 Advantages of the ATD

The ATD gives the user of AQL a free computational workflow. With few exceptions, the user need not supply complete typing information currently with the query commands. Algebraic statements in AQL are a way to propose entity relationships, which are then tested or validated by execution. The AQL user need not define the form of their result before making a query. In this sense, AQL can be thought of functionally as a database calculator, allowing direct computation without predicting outcomes, declaring tables, or using functions that are not directly related to a simple query. This notion of "AQL as calculator" is further supported by the inherent recursion supported in AQL expressions, and by their arbitrary length and complexity.

The closure implied in Codd's relational model enforced not just a relational structure, but a coherence of information. He felt that even to be called a relational operator, it must provide inferences that fit into a schema of design, hence the vital role of the ERD in the use of SQL. This could be thought of as strongly “relation-centric”. In AQL, the ATD allows users to find inferences that are not predefined in that sense, to make novel use of operators to discover unexpected results without having to manage burdensome typing details. This is another way in which AQL is said to be “operator-centric” and have an open workflow.

7 *AQL at JPL*

AQL was originally written by Marc Kidwell Pestana to assist with the GPS Earth Observatory (GEO) group at JPL, and to track GEO occultation data through the GEO Occultation Analysis System (GOAS) [5].

The GEO program is involved in climate, weather, and ionospheric research. Here, the global positioning system (GPS) is exploited via radio occultation techniques to obtain profiles of refractivity, temperature, pressure, and water vapor in the neutral atmosphere and electron density in the ionosphere. These profiles are called "occultations". A single spacecraft can generate up to 600-700 occultations in a day. In processing these signals, care must be taken to separate the numerous factors that can affect the occulted signal. These include the motion of the satellites, clock drifts, relativistic effects, the separation of the ionosphere and the neutral atmosphere, and the contribution of the upper atmosphere where the sensitivity of the GPS signal is weak [13].

This program generates an enormous variety of atmospheric models and data products. Over 150 databases exist in the occultation system. The total occultation database is close to 4 TB in size. Scripts are embedded in six locations within the system that write new data records to the GEO Database Interface. AQL is then used to create and manage occultation databases that track the profile processing results throughout the system and to determine the reason for any deviations that may arise from atmospheric data sets generated by other data centers. Satellite, latitude, and longitude of occultation, computational strategy,

date, start time, end time, status, failure mode, and the occultation reference satellite are among the parameters involved, along with several others.
In this environment, AQL is used for:

- Co-locating occultations from different satellites based on latitude and longitude positioning and epoch
- Diagnosing system failures
- Throughput comparison of the GOAS system against the UCAR system.
- World distribution map of occultations over a $5^{\circ} x 5^{\circ}$ grid
- Database administration: 8 – 24 million transactions updating every record in the GEO Database performed in less than a day while maintaining data integrity constraints (this is a prime example of rapid database migration)

8 *Forward into Matrix Algebra, Machine Learning and A.I.*

Due to the similarities between relations, matrices, and tensors, AQL has been recently expanded to include matrix and tensor algebra. This allows search and computation on matrices, tensors, and relations in the same expression, opening an exciting path for future research and development. New smart function operators can allow for experimentation and computation with machine learning algorithms and AI models. Combining machine learning and inferential query throughout the pipeline could have a major impact on the development of machine intelligence.

It has been noted that AQL was developed firstly for scientific applications. This may lead to the conclusion that AQL could only be used in this way. However, AQL is not in any way restricted to a particular class of data. The ADBMS has the same mathematical foundation as SQL: the relation and relational algebra. AQL uses relations to group facts into units of knowledge, be they scientific facts, financial facts, or otherwise. Both models are agnostic as to the type of data they store. The omission of some of Codd's data integrity constraints in the current implementation of AQL was a choice made by the developer, not an inherent limitation of the AQL database model. Those constraints were omitted to provide for maximum ease of use and easy access to data, but they could be restored if needed. These integrity constraints are access controls, referential integrity, and business rules.

Although development is pending, it seems to be most practical to use the ATD as a starting point for the implementation of integrity constraints. The ATD already enforces three of Codd's integrity constraints. The AQL operators use the ATD to check the validity of each operation being executed, therefore the other two constraints can be checked in a similar way. Using the ATD has the additional benefit of maintaining the AQL workflow and operator centrism.

For example, access constraints could be made a part of the datatype of each relation in the database. Access control could be managed by the ATD by extending the type of each relation with permissions, providing the designer with more granular control.

Second, referential integrity constraints could be implemented in each operator via the ATD. Whether or not to enforce referential integrity between two relations would be designated in their types. Thus, the ATD would control the enforcement of referential integrity at the level of each AQL operator.

Third, business rules could be enforced with AQL expressions interpreted as constraints. Imagine the following constrain operator expressed in AQL pseudocode:

```
constrain( (A  &&  B)  ::  @a < 100 & @b .eq. “Mary” )
```

This operator would take an AQL expression as a single argument. The argument shown above is an inner join (or theta join) of A and B between all records from table A where the attribute ‘a’ is less than 100, and records from table B where the attribute b is equal to the name “Mary”. However, interpreted as a condition, it disallows any insertion or deletion that would result in satisfying the predicate.

Moreover, the following expression would enforce the constraint for the negation of this condition:

```
constrain( (A  &&  B)  ::  !( @a < 100 & @b .eq. “Mary”) )
```

Here the not operator “!” would negate the condition. Again, the ATD would check the validity of each operation. Thus, business rules could be expressed using the same language used for search expressions, only interpreting them as constraints, while still maintaining an open workflow.

While the imposition of Codd’s data integrity constraints might slow down data access in any DBMS due to increased overhead, they don’t necessarily preclude AQL’s open workflow. In fact, making these constraints part of the datatype of each relation in the system permits a mixture of relations both under and not under integrity control. Thus, open workflow and open architecture apply to integrity constraints as well.

To expand on this concept, we could add constraints beyond Codd’s integrity constraints. They can be anything for any reason, so long as they can be written as valid AQL expressions. The use of a constrain operator in deep learning might aid in the training process.

9 *Conclusion*

In this paper we have briefly covered notable aspects of Algebraic Query Language (AQL) by looking at the background of relational database management, contrasting AQL’s model, purpose and use with SQL’s, providing a formal AQL definition with use cases, discussing its innovative Automatic Type Database (ATD), the ramifications to user workflow and a look at current and future developments in search and machine intelligence and business applications. We hope that AQL can be made available to a wide variety of users, not just scientists and engineers. AQL will soon be available in the cloud to use on a trial basis for prospective users to evaluate. Please contact the Caltech Office of Technology Transfer website: https://ott.jpl.nasa.gov email: ott@jpl.nasa.gov for licensing information, and the author Marc Kidwell Pestana at Algebraisk, LLC website: http://www.algebraisk.com email: scient@algebraisk.com for support and research assistance.

10 *Acknowledgements*

We would like to give a thank you to the following people and organizations for their support in helping us develop AQL, the generosity of sharing their expertise, and their help in getting this whitepaper done:

NASA
CalTech
Jet Propulsion Laboratory
Dr. Richard Gross
Tony Mannucci
Olga Verkhoglyadova
Kevin Mitchell
Marvin Perlman
Michael Rael
Jason Moore
Valerie Hinton

And a special thanks to Joseph Ledette. Without your support, I could never have done this. - MKP

11 Citations

[1] L. Chao, "Database Development and Management," in *Database Development and Management*, 1st Edition ed., T. &. F. Group, Ed., Boca Raton, Florida: Auerback Publications, 2006.

[2] M. L. Scott, "Programming Language Pragmatics," in *Programming Language Pragmatics*, 3d Edition ed., Burlington, Massachusetts: Morgan Kaufmann Publishers, 2009.

[3] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM,* vol. 13, no. 6, p. 377–387, June 1970.

[4] E. F. Codd, "The Relational Model For Database Management," in *The Relational Model For Database Management*, vol. 2, Menlo Park, California: Addison-Wesley, 1990, p. 538.

[5] G. A. Hajj, "CHAMP and SAC-C atmospheric occultation results and intercomparisons," *Journal of Geophysical Research,* vol. 109, p. 24, 29 October 2004.

[6] D. D. Chamberlin, "Early History of SQL," *IEEE Annals of the History of Computing,* pp. 78-82, October-December 2012.

[7] E. F. Codd, "A Data Base Sublanguage Founded on the Relational Calculus"," in *Proceedings of the ACM SIGFIDET Workshop on Data Description, Access, and Control*, 1971.

[8] D. D. Chamberlin, "SEQUEL: A Structured English Query Language," in *SIGFIDET '74: Proceedings of the 1974 ACM SIGFIDET*

(now SIGMOD) workshop on Data description, access and control, 1974.

[9] E. F. Codd, "Relational Completeness of the Data Base Sublanguages," in *Courant Computer Science Symposia 6 Data Base Systems*, San Jose, California: Prentice Hall, 1972, p. RJ 987 (#17041).

[10] L. De Hann and T. Koppelaars, "Database Skeleton," in *Applied Mathematics fro Database Professionals*, 1st Edition ed., J. Gennick, Ed., New York, New York: Apress, 2006, pp. 144-147.

[11] Akmajian, Demers, Farmer and Harnish, Syntax, Cambridge, Massachusetts: The MIT Press, 1955, pp. 192-3.

[12] M. Perlman, "Abstract Algebra," in *Abstract Algebra*, La Canada, California.

[13] "University Corporation for Atmospheric Research," [Online]. Available: http://www2.ucar.edu.