The Bloor Group

DOING THE MATH

The Algebraix® DataBase: What it is, how it works, why we need it

WHITE PAPER

Executive Summary

Algebraix Data recently introduced a highly unusual BI database product, named A²DB. The database is the realization of a new and highly versatile ***Algebra of Data***, which is directly derived from classical set theory. As far as we can tell, it is valid, viable and versatile. As this white paper will explain, this unique algebra is the foundation of the A²DB database, and is the reason for all of the positive characteristics of the product. This new algebra subsumes the relational algebra created by Tedd Codd in the 1970s, and hence is able to replace it.

Detailed explanations of how and why A²DB provides the capabilities it does are found in this paper. The following is a list of the benefits that A²DB delivers or claims to deliver:

- **Low hardware costs.** A²DB runs on commodity x86 hardware. It is economic in its approach. Some small to medium size BI applications that might otherwise require an MPP configuration, will fit into SMP with A²DB.
- **Scalability:** A²DB scales well (roughly linearly) within SMP environments and, in such environments, will accommodate data growth without performance issues arising. It does not yet have MPP capability.
- **Extremely low operational overhead:** A²DB is self-managing in many ways that other databases are not. There is no need to manage resource space allocated. Data take-on and update is simple. Fewer database instances are required, both for physical reasons and for design reasons. In particular there is no requirement for DBAs to tune the database for performance as it is self-tuning.
- **Database design effort reduces almost to zero:** A²DB is extremely forgiving in respect of database design. Any design that is consistent in respect of data is a viable starting point. The algebraic nature of the database compensates for imperfect data design.
- **Improved performance and accelerating performance:** A²DB delivers high performance; and, uniquely among database products, the performance improves with usage. It is particularly suited to BI applications in the 1- 2 terabyte arena where performance has been an issue for other database products.
- **Faster time to value:** Algebraix Data claims A²DB can deliver a faster time to value than competitive products. This stems from several attributes of the product, including its fast load times, self-tuning capability and the fact that a single instance can be used both as a data warehouse and a collection of data marts.
- **Capable of handling difficult BI applications:** Because of A²DB's algebraic management of data, it is better able to handle BI than traditional applications that have historically been regarded as difficult. A²DB handles dynamic metadata, sparse data, and is particularly suited to regulatory applications because it holds a full audit trail of all changes, and it enables point-in-time queries.
- **Strategic foundation:** A²DB will work on some BI applications that have previously proved difficult to implement. Additionally, Algebraix Data claims that, because A²DB is so versatile, it can occupy a foundational position within any BI architecture. It reduces ETL effort, eliminates DBA work, manages metadata changes. In general, BI applications that were previously impossible have now become possible, thus benefits not previously delivered now become achievable.

Mathematics In the Aftermath

Successful inventors are usually engineers, not mathematicians. They experiment with various designs, produce something that works and unleash it on the world. That's what the Wright brothers, Thomas Edison and others like them did. They didn't agonize over mathematical formulae and ponder differential equations; they built prototypes, discarded the ones that didn't work and improved the ones that did. The mathematicians arrived in the aftermath, long after such inventions were put into service, applying their mathematical skills to enhance and improve the brainchildren of the engineers.

Consider the cannon. It was invented by the Chinese in the 12th century and knowledge of it spread to Europe. By the 15th century engineers were building very effective cannons. But it wasn't until the late 17th century, when Isaac Newton published has *Principia Mathematica* that anyone actually had the mathematics to calculate a cannon ball's trajectory. In the centuries that followed mathematicians were gainfully employed calculating artillery tables. Indeed the development of electronic calculators in the USA during World War II was driven by the need to calculate artillery tables. The invention came first and the math came afterwards.

The exception to this rule was the computer itself. It was invented by mathematicians, who chose to become engineers and collaborate with engineers. Take a bow; Charles Babbage, Alan Turing and John Von Neumann. But after the original conception and early prototypes, mathematicians had little to do with the computer's further evolution. In the universities, Computer Science split off as a completely separate discipline and mathematicians paid little further attention to their world-shaking invention.

After the 1950s, there was only one area where Mathematics and Computer Science intersected in a significant way. This was in the formulation of the relational model of data, which first saw the light of day in Ted Codd's landmark paper; ***A Relational Model of Data for Large Shared Data Banks.*** The impact that this had was profound. Within a decade many entrepreneurial companies were building databases that tried to conform to the relational model that Ted Codd had defined.

The Incompleteness of the Relational Model of Data

It was known from the beginning that the Relational Model of Data was incomplete. There was some theorizing and academic chatter around this topic, but the imperfection of the model became obvious primarily because of the difficulties that were experienced when using relational database (RDBMS) with particular data structures. RDBMS were fine when data naturally fit into tables. However they had problems in accommodating data that was hierarchically structured. In the early days, RDBMS were particularly challenged by Bill Of Materials Processing (BOMP).

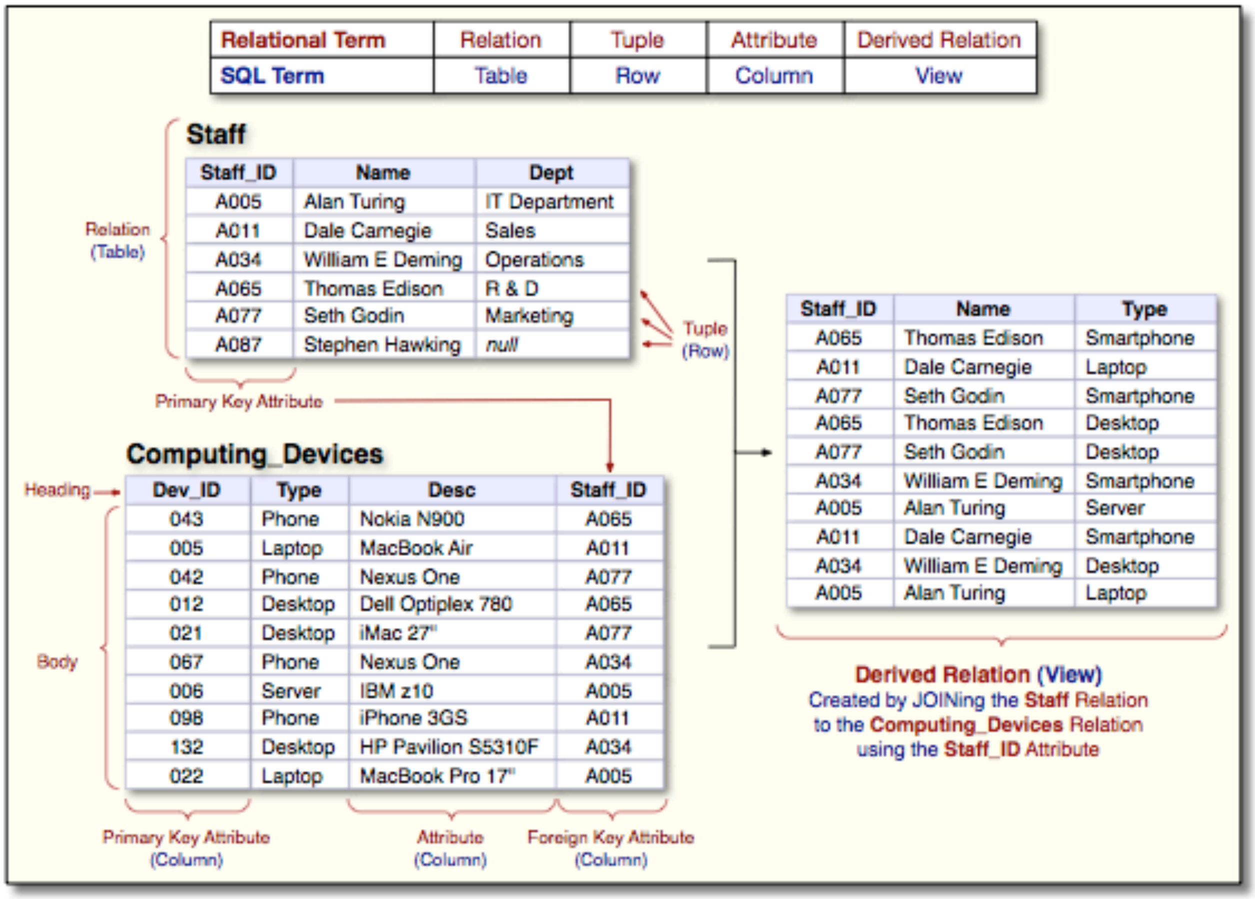
Later on, when Object Oriented Programming (OOP) came on the scene, the famous "impedance mismatch" emerged. There were complexities to the situation, but in simple terms, the problem was this: ***The way that OOP languages wanted to handle data conflicted with the way RDBMS wanted to manage data.***

Since OOP languages quickly became dominant among programmers and RDBMS were already dominant as databases, the solution that prevailed was a compromise. It involved having a third component, Object Relational Mapping (ORM) software, that translated from the object view to the relational view for the purpose of data storage. That sounds inconvenient, and that's because it is inconvenient, but it works.

It is clear from this that the Relational Model of data was incomplete, but that was never really in dispute. If the Relational Model had been comprehensive enough, programmers and designers would have used it as a standard for data storage for every new kind of application. But they never did.

However, maybe now they have something to work with. Because now, decades after the dawn of Relational Database, a comprehensive algebraic model of data has been created and used to build a database product. The mathematicians are back.

The Relational Model: A Brief Description



Relational Term	Relation	Tuple	Attribute	Derived Relation
SQL Term	Table	Row	Column	View

Staff

Staff_ID	Name	Dept
A005	Alan Turing	IT Department
A011	Dale Carnegie	Sales
A034	William E Deming	Operations
A065	Thomas Edison	R & D
A077	Seth Godin	Marketing
A087	Stephen Hawking	*null*

Computing_Devices

Dev_ID	Type	Desc	Staff_ID
043	Phone	Nokia N900	A065
005	Laptop	MacBook Air	A011
042	Phone	Nexus One	A077
012	Desktop	Dell Optiplex 780	A065
021	Desktop	iMac 27"	A077
067	Phone	Nexus One	A034
006	Server	IBM z10	A005
098	Phone	iPhone 3GS	A011
132	Desktop	HP Pavilion S5310F	A034
022	Laptop	MacBook Pro 17"	A005

Staff_ID	Name	Type
A065	Thomas Edison	Smartphone
A011	Dale Carnegie	Laptop
A077	Seth Godin	Smartphone
A065	Thomas Edison	Desktop
A077	Seth Godin	Desktop
A034	William E Deming	Smartphone
A005	Alan Turing	Server
A011	Dale Carnegie	Smartphone
A034	William E Deming	Desktop
A005	Alan Turing	Laptop

Relation (Table)
Tuple (Row)
Primary Key Attribute
Heading
Body
Primary Key Attribute (Column)
Attribute (Column)
Foreign Key Attribute (Column)
Derived Relation (View)
Created by JOINing the Staff Relation to the Computing_Devices Relation using the Staff_ID Attribute

Figure 1. The Basics of the Relational Model

It will be easier to appreciate how the Algebraic Model of Data differs from the Relational Model, if we describe the Relational Model and its terminology using a simple example, and then use the same example to illustrate the Algebraic Model. Figure 1. illustrates the basic terminology of the Relational Model.

Discussions about the Relational Model are sometimes confused by the fact that there are two sets of terminology in play. It was originally described using the terms shown at the top of the illustration in red; Relation, Tuple, Attribute and Derived Relation. However once SQL became established as the de facto access language, the terms Table (Relation), Row (Tuple), Column (Attribute) and View (Derived Relation) became common. The terms are equivalent, as the illustration shows.

It is important to note that we are discussing a ***logical*** model here. In arranging data into tables, the Relational Model ***logically*** represents a collection of data, making no assumption at all about how that data might be ***physically*** represented when stored on a disk or in memory. The model arranges data in distinct ***logical*** tables, which directly correspond to real-world things (or ***entities***) like Staff and Computing Devices, as in the example.

This is clearly sensible. ***Entities*** are nouns and ***attributes*** are qualifiers of those nouns. Data about collections of ***entities*** can be arranged in ***relations*** (tables), where the heading describes the ***attributes*** of the ***relation***. Incidentally, there is no explicit or implied sequence to the ***tuples*** in a ***relation*** (table). The ***relation*** is unordered by definition.

Also note the appearance of a null entry in the tuple with Staff_ID: A087 and Name: Stephen Hawking. This tuple might represent, for example, that we have hired Stephen Hawking but have yet to assign him to a given Department. In the Relational Model, if there is no value for an attribute in an individual row, a ***null*** is inserted. It is a marker indicating that there is no value. The use of nulls can cause headaches, so some designers prefer not to allow them. However, designing around them can cause headaches too.

An ***attribute*** can also be a key. If it is a ***primary key,*** it is required to have a unique value. Thus the ***primary key*** uniquely identifies the ***tuples*** in the ***relation***. A unique ordering of the ***relation*** (table) can easily be achieved by sorting the ***relation*** in order by its ***primary key***. An ***attribute*** can also be a ***foreign key*** (as the Staff_ID is in the illustration), in which case it naturally forms a link between two ***relations*** (tables). This makes it possible to create a ***derived relation*** (view) by joining the two original ***relations*** using the Staff_ID.

The strength of the relational model lies in the fact that a good deal of the data that we need to process and analyze fits neatly into tables and thus databases organized around relational principles have a fairly wide area of application.

The Algebraic Model of Data

The Algebraic Model of Data that sits beneath Algebraix's A²DB does not force data into two-dimensional tables. Instead it represents data using the more familiar mathematical structure of sets. This turns out to be a more flexible structure for data. To get a sense of this algebra, you need to understand four simple ideas:

1. The ***couplet***
2. The ***extended set***
3. The ***clan***
4. The ***derived clan***

These ideas are illustrated in Figure 2. (on the following page). They employ the same example of data we used when illustrating the Relational Model. However, we are now dealing with sets and not tables.

In this model, every datum is a ***couplet***; {s,v}, consisting of a scope (the metadata) and the datum's value. These ***couplets*** can be collected together into ***extended sets***, which consist of

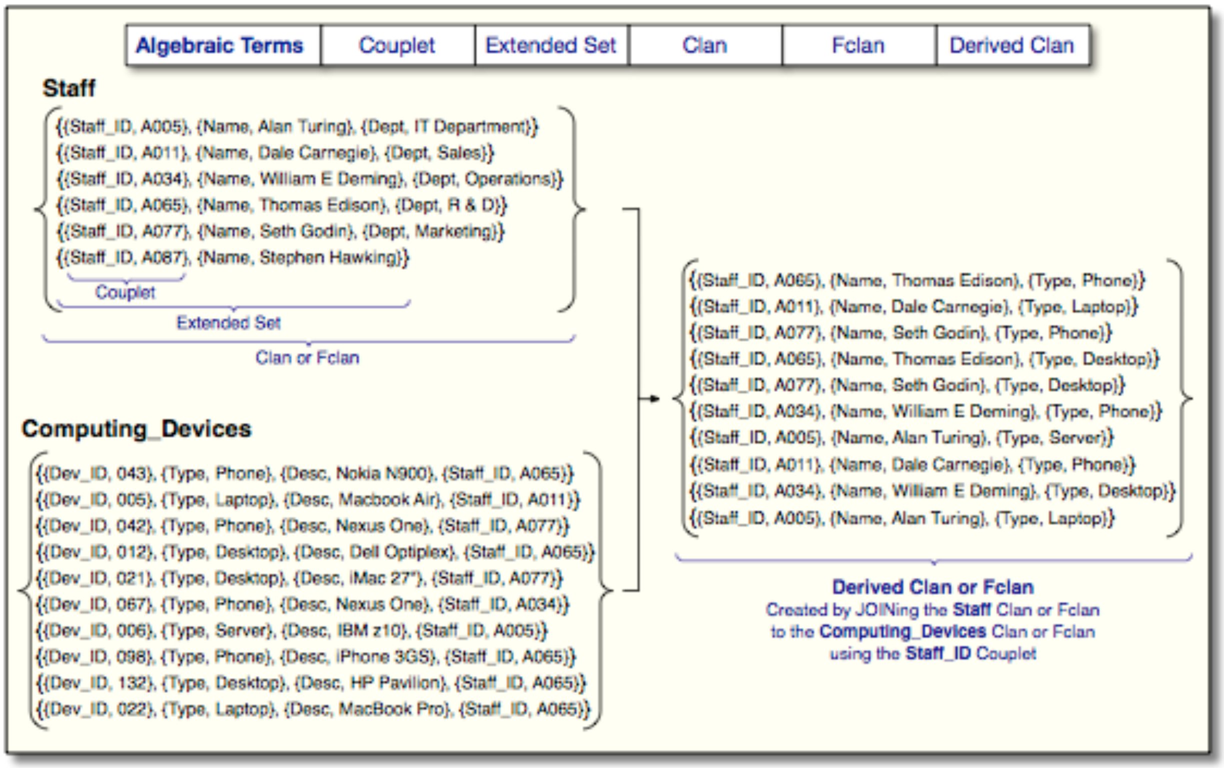

Algebraic Terms
Couplet
Extended Set
Clan
Fclan
Derived Clan
Staff
{{Staff_ID, A005}, {Name, Alan Turing}, {Dept, IT Department}}
{{Staff_ID, A011}, {Name, Dale Carnegie}, {Dept, Sales}}
{{Staff_ID, A034}, {Name, William E Deming}, {Dept, Operations}}
{{Staff_ID, A065}, {Name, Thomas Edison}, {Dept, R & D}}
{{Staff_ID, A077}, {Name, Seth Godin}, {Dept, Marketing}}
{{Staff_ID, A087}, {Name, Stephen Hawking}}
Couplet
Extended Set
Clan or Fclan
Computing_Devices
{{Dev_ID, 043}, {Type, Phone}, {Desc, Nokia N900}, {Staff_ID, A065}}
{{Dev_ID, 005}, {Type, Laptop}, {Desc, Macbook Air}, {Staff_ID, A011}}
{{Dev_ID, 042}, {Type, Phone}, {Desc, Nexus One}, {Staff_ID, A077}}
{{Dev_ID, 012}, {Type, Desktop}, {Desc, Dell Optiplex}, {Staff_ID, A065}}
{{Dev_ID, 021}, {Type, Desktop}, {Desc, iMac 27"}, {Staff_ID, A077}}
{{Dev_ID, 067}, {Type, Phone}, {Desc, Nexus One}, {Staff_ID, A034}}
{{Dev_ID, 006}, {Type, Server}, {Desc, IBM z10}, {Staff_ID, A005}}
{{Dev_ID, 098}, {Type, Phone}, {Desc, iPhone 3GS}, {Staff_ID, A065}}
{{Dev_ID, 132}, {Type, Desktop}, {Desc, HP Pavilion}, {Staff_ID, A065}}
{{Dev_ID, 022}, {Type, Laptop}, {Desc, MacBook Pro}, {Staff_ID, A065}}
{{Staff_ID, A065}, {Name, Thomas Edison}, {Type, Phone}}
{{Staff_ID, A011}, {Name, Dale Carnegie}, {Type, Laptop}}
{{Staff_ID, A077}, {Name, Seth Godin}, {Type, Phone}}
{{Staff_ID, A065}, {Name, Thomas Edison}, {Type, Desktop}}
{{Staff_ID, A077}, {Name, Seth Godin}, {Type, Desktop}}
{{Staff_ID, A034}, {Name, William E Deming}, {Type, Phone}}
{{Staff_ID, A005}, {Name, Alan Turing}, {Type, Server}}
{{Staff_ID, A011}, {Name, Dale Carnegie}, {Type, Phone}}
{{Staff_ID, A034}, {Name, William E Deming}, {Type, Desktop}}
{{Staff_ID, A005}, {Name, Alan Turing}, {Type, Laptop}}
Derived Clan or Fclan
Created by JOINing the Staff Clan or Fclan
to the Computing_Devices Clan or Fclan
using the Staff_ID Couplet

Figure 2. The Basics of the Algebraic Model

one or more such ***couplets***, as illustrated. The extended sets can be gathered into ***clans***, as also illustrated. We can also create ***derived clans*** by, for example, a JOIN operation using the Staff_ID ***couplet*** to bring together ***couplets*** from the ***extended sets*** in the Staff ***clan*** and Computing_Devices ***clan***.

The illustration may be a little misleading due to our deliberately drawing a parallel between ***relations*** and ***clans***. Algebraically, there are no constraints on any of the collections. There are no rules insisting that, for example, each ***couplet*** in an ***extended set*** must have different metadata from every other ***couplet***. Neither is there any rule saying that all ***extended sets*** within a ***clan*** must have exactly the same number of ***couplets*** or even that each ***extended set*** must have a ***couplet*** that equates to a ***primary key***. In fact there is no notion of a key anywhere in the Algebraic Model of Data.

This may appear dangerously unconstrained, but it isn't. So far we have not discussed ***fclans*** and it is the ***fclans*** that make the Algebra of Data practical. An ***fclan*** is a ***clan*** to which a function or constraint is applied.

For example, if we apply a rule to the Staff ***clan*** which says that the ***clan*** only allows the inclusion of ***extended sets*** which consist of a Staff_ID ***couplet***, Name ***couplet*** and optionally a Dept ***couplet***, then we have an ***fclan*** that has a very definite structure. You may have noticed, from the illustration, that the Algebraic Model does not enforce a table-like structure. Consequently, it does not force ***couplets*** with ***null*** values into a ***clan*** or an ***fclan***. However, in an ***fclan***, it is possible to add that constraint. It is also possible to add the constraint that the Staff_ID ***couplet*** must be unique between all ***extended sets***. If we do all of these things, we get direct equivalence to a ***relation***.

We have now described enough about how the Algebraic Model of Data works to be able to describe how the BI database A²DB works. Hopefully, you will have got the impression that the Algebraic Model of Data is capable of subsuming the Relational Model. That is exactly the case. It can represent ***relations***, ***attributes***, ***tuples***, ***primary keys*** and ***foreign keys***. The underlying algebra is also capable of representing any of the operations that are performed on ***relations***. So any ***derived relation*** that can be created using SQL, for example, can be arrived at by employing the algebra.

Where's the Beef?

The power of the Algebraic Model is that it can represent both the ***logical*** AND the ***physical*** representations of data in a database. To clarify this, it helps if we define what we mean by logical and physical:

Logical Model: The logical data model is the data structure that the programmer or IT user refers to in making requests for data. Within the Relational paradigm, the user thinks in terms of tables with rows and columns.

Physical Model: The physical data model is the data structure employed for storing data. There are many different ways that data can be stored in order to enable efficient retrieval.

The basic principles of a database are simple in respect of ***logical*** and ***physical*** data models. They can be summarized as follows:

1. The ***logical*** model organizes data in a way that is easy to understand and easy to access using a programing language or query language.
2. Neither the IT user, nor the programmer should have any need to know how data is physically stored. They should only need to understand the ***logical*** model of data.
3. The ***physical*** model organizes the data in such a way as to optimize its speed of access while ensuring effective data management.
4. A well designed database is far better able to organize data for optimal performance than any programmer. It can know far more about the data access activity it experiences and it can know how best to store the data.
5. It is the role of the database to accept data access calls made to the ***logical*** model, translate those calls into requests to the ***physical*** model and optimize the retrieval of data accordingly.

That's database in a nutshell - or it should be. Unfortunately relational databases don't implement principles 3, 4 and 5 particularly well. The Relational Model is out of its depth trying to represent the physical on-disk and in-memory data structures that are likely to optimize the retrieval of data. Quite simply; it cannot and so it doesn't. As a consequence, relational databases use a grab bag of engineering tricks and techniques in their efforts to construct and manage efficient physical data stores.

There are many such techniques, but they tend to boil down to five distinct variations: Data partitioning, Data compression, Indexing, Parallel processing and Hardware acceleration (using purpose built processors).

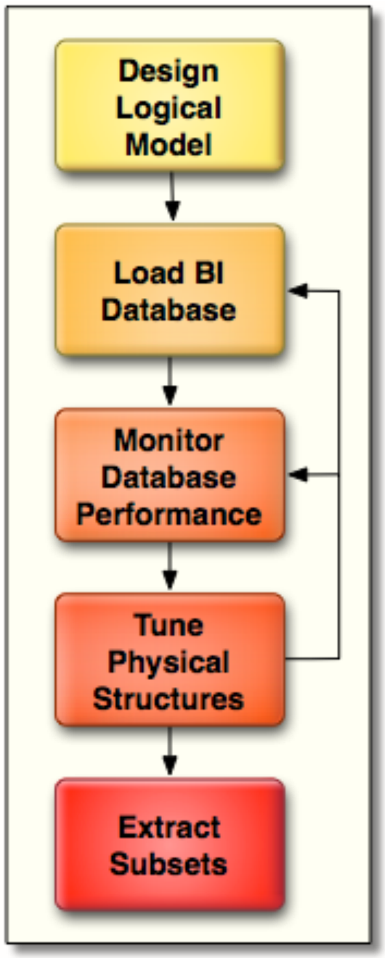
In the BI market, where data volumes have grown from gigabytes to petabytes in the past two decades, there has been a procession of new database products - prompted primarily by the sluggishness of the popular RDBMS in BI applications. Many of the new products are based on the "column store" approach, first implemented by Sybase IQ. The idea is to store very large tables using vertical partitioning, splitting such tables into columns, and even horizontally partitioning the columns across different disks or even different servers - for the sake of parallel processing.

Others, including products from Netezza and Kickfire, bet on hardware acceleration. Still others like GreenPlum are pushing the envelope on scale-out capabilities across parallel grids of commodity servers using partitioning for the sake of speed. There are even the so-called NoSQL products, which "laugh in the general direction" of RDBMS. Their primary approach is to avoid JOIN operations via amalgamating tables together and applying extreme parallelism. Implementations include Google's BigTable and Amazon's Dynamo.

The fact that there are many products using a bewildering variety of ***physical*** tactics to deliver BI performance only confirms that there is no algebraic approach being applied to the physical storage of data - or there wasn't until A²DB came on the scene.

The Accepted BI Process

Figure 3. Shows a typical set of activities carried out in implementing a data warehouse or an operational data store (ODS). There are other activities that may be involved, such as ETL (Extract, Transform, Load) and Master Data Management (to rationalize disparate data models and contexts), but we are focusing here on BI database design and performance.



The first activity, ***Design Logical Model*** involves designing the user's view of the data. Whole books have been written about this and further ones could be. To discuss this topic in any depth requires some discussion of Master Data Management, since that activity involves building a data model that fits the whole organization. However, such things are often peripheral to this activity, because the so-called ***logical*** model that developers build will be driven more by the physical database they are using than anything else. In fact, it wont really be a ***logical model*** at all.

Not long after the initial enthusiasm for data warehousing took root, the idea emerged that, in many data warehouses the ideal data structure was a "star schema" or a "snowflake schema." A star schema involves a database design consisting of just one very large "Fact Table" and several other related tables. Following fast on its heels came another popular general design idea, the snowflake schema, which had a few more tables and was a little more complex and hence the design resembled a snowflake rather than a star.

Decades earlier, in 1971, the idea of database normalization had been introduced and it was once believed that a database normalized to 3rd normal form was the "correct" logical design.

However, such designs, when implemented, ran far too slowly to be practical. Additionally, while a 3rd normal form data model might have some mathematical virtues, it was rarely a design that IT users found easy to comprehend.

The reality was that the logical model design is best done by someone who understood how the database worked and how it was likely to be used, so that a schema could be created that would be comprehensible to database users and would take advantage of the performance characteristics of the database it rode on.

We are drawing attention to the ***Load BI Database*** process simply because it is time-consuming from a computing perspective. How long it takes to initially load the database naturally depends upon the product being used, and the tactics it uses to deliver acceptable performance. However, when a database is very, very large, reloading it can be hugely expensive. So the penalty for a bad design - one that is so bad that it needs to be fundamentally changed - is very high.

Now consider the ***Monitor Database Performance*** activity. The reality is that nobody knows whether a given "logical design" is good until users run queries. The proof of the pudding is in the eating. Naturally users want the fastest response they can possibly get and, in strict business terms slow responses are expensive. So the Database Administrators (DBAs) use the capabilities the database provides to track performance.

The ***Tune Physical Structures*** activity is what naturally follows performance monitoring. There will normally be some parameters that can be altered or some tactics that can be employed for situations where some queries run slow. Good DBAs have sufficient experience of a database to know how to tune it up when it performs poorly.

However, if nothing can be done at the physical level to improve performance, then in many situations it's possible to ***Extract a Subset*** of the data for a given set of users and set them up with their own dedicated database capability. It isn't always possible, because some users may well need to be able to access all the data all the time, but it is a common tactic that is used when all else fails.

Of course, OLAP databases are a special example of this. Not only do they hold data subsets pulled from a data warehouse, but they store data in a way that facilitates queries in multiple dimensions - a typical example is sales reports, where the user wishes to drill down into sales by region, and by customer, and by product, over given time frames. The typical data warehouse does not handle such requirements well, but an OLAP database lives for such queries.

A Database of a Different Kind

Algebraix Data claims that the BI activities just described are either unnecessary with A^{2}DB or far simpler. In particular, the database is self-tuning, data-loading activities are much simpler, and designing the logical model requires far less effort. To understand why this may be, we need to appreciate how A^{2}DB works, and we need to understand its foundational algebra of data management.

The Advanced Analytical Database, A²DB

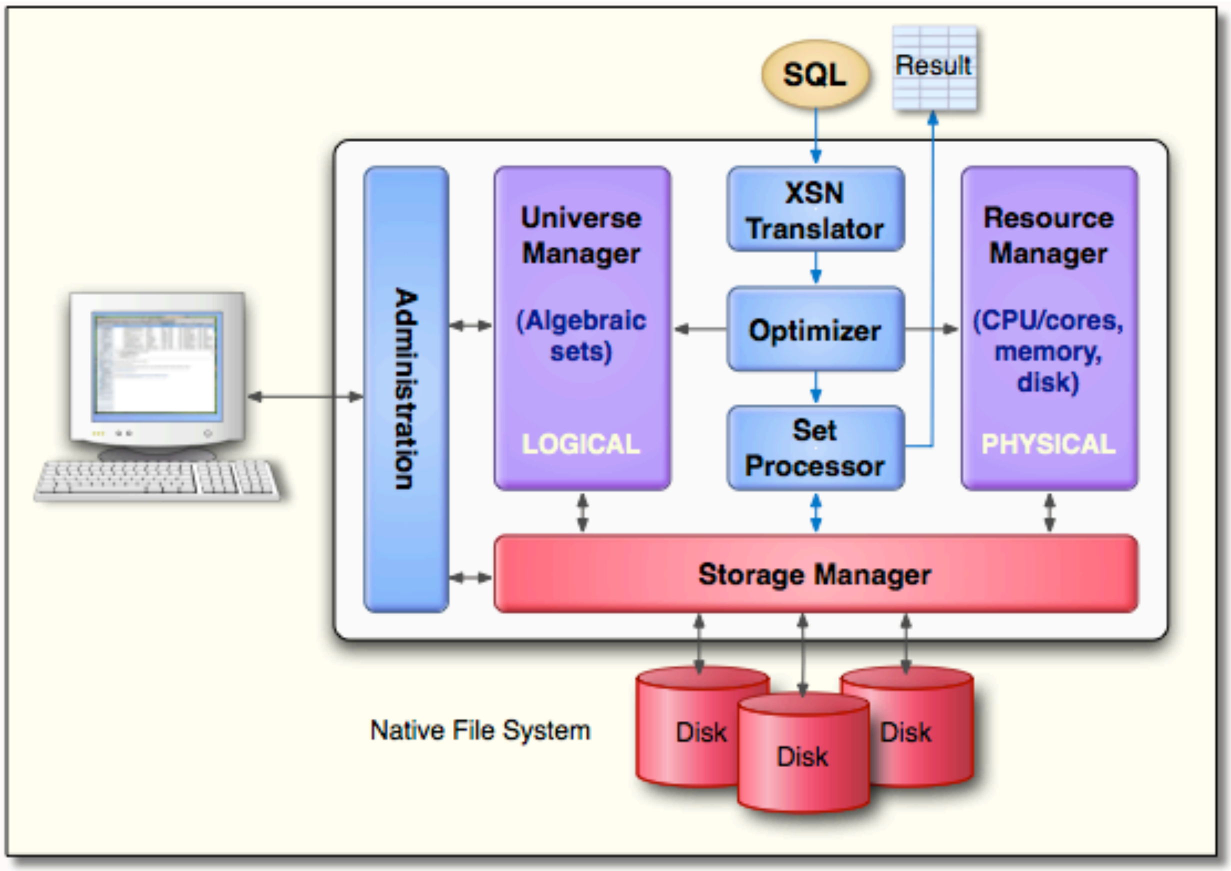


Figure 4. The Advanced Analytic Database

Architecturally, A²DB is unusual. It doesn't employ a proprietary file structure. As indicated in Figure 4. it uses the native file system (whatever it is) for all files. And, remarkably, all data is stored (initially) in delimited files with no indexes (that's right, CSV files - and no indexes).

If you want to load new data into an A²DB database, you extract a CSV file from the data source and write it to the A²DB server. Aside from that, loading data only involves telling A²DB what the metadata structure of the CSV file is and where it is on disk. If the file has some fields in common with any of the other data in the A²DB database (like a Customer ID or whatever) this is deduced from the queries that users create.

In effect, loading data reduces to loading metadata. It takes seconds. Data is loaded and bound to a structure with the CREATE TABLE or INSERT statement via the SQL interface through a proprietary extension.

A²DB has a SQL interface based on the SQL92 standard. It uses the Data Direct connection software (from Progress Software) to support other SQL dialects including JDBC and Microsoft's ADO .NET. Data Direct translates these into A²DB's preferred dialect of SQL. Other interfaces are possible, although no other interfaces have been built yet.

Logical and Physical

As Figure 4. suggests, A²DB implements a complete separation between **LOGICAL** and **PHYSICAL**. This is the key to understanding how the product works. This is what it does:

The **XSN Translator** translates a SQL query into an algebraic representation that corresponds with the algebraic sets defined in the **Universe Manager**. (**XSN** stands for Extended Set Notation, which is used to create an algebraic representation of a query). The **Universe Manager** holds a complete **LOGICAL** model of all A²DB sets and their relations, including those clan sets which correspond to relational tables.

Mathematically, a query is simply a **function** applied to a **clan**. That's what it is. You might think of it as, say, "**SELECT** *Name, Tel_No* **FROM** *Staff* **WHERE** *Gender = 'F'* **ORDER BY** *Name*", but in algebraic terms that is a **function** applied to the Staff **clan**. So the **XSN Translator** translates the SQL into an algebraic **function** and passes the **function** to the **Optimizer.**

The **Optimizer** generates equivalent expressions to the **function** through algebraic manipulation based on the properties of the algebra and the known universe of sets and expressions. This will yield one or more possible solutions. Later, we will explain why there can be more than one solution, even in apparently simple situations, but for the moment just assume that it is so.

Having done the **LOGICAL** work, the **Optimizer** consults the **Resource Manager** and it tests each of its algebraic solutions against **PHYSICAL** information held by the **Resource Manager**. The **Resource Manager** is like my Scottish uncle who always knows the cost of everything. Armed with precise cost information, the **Optimizer** works out the **PHYSICAL** cost of each algebraic solution and chooses the fastest one.

The **Resource Manager** knows the costs because it knows where all the **PHYSICAL clans** are *and how they are physically organized*. Some may be on disk. Some may be in memory. It knows how much space they occupy and how fast they can be accessed.

You may be thinking that there is not much difference between the **LOGICAL** and **PHYSICAL** here, but there is. The **LOGICAL** map maintained by the **Universe Manager** is an accurate algebraic expression of a collection of clans. The **PHYSICAL** storage map transforms the **LOGICAL** map, doing neat time-saving tricks that other databases do, like compressing data or partitioning files. The **PHYSICAL** map owned by the **Resource Manager** identifies where the data is and how it is stored.

Ultimately, the **PHYSICAL** storage of a **clan** is just another mathematical **function** applied to a **clan**. The **LOGICAL** optimization is simply algebra, and conveniently, so is the **PHYSICAL** optimization. So the **Optimizer** solves some equations and passes the fastest solution to the **Set Processor,** which it executes.

The Set Processor: What it is and what it does

The **Set Processor** follows a plan of action. Each step it takes creates a new **clan** of some kind. We illustrate this idea very simply in the Figure 5. (next page):

For visual simplicity (only) we're illustrating this with tables. We have two **clans**, **A** and **B**. The **extended sets** of these two **clans** have one metadata value in common. So **A** might be a

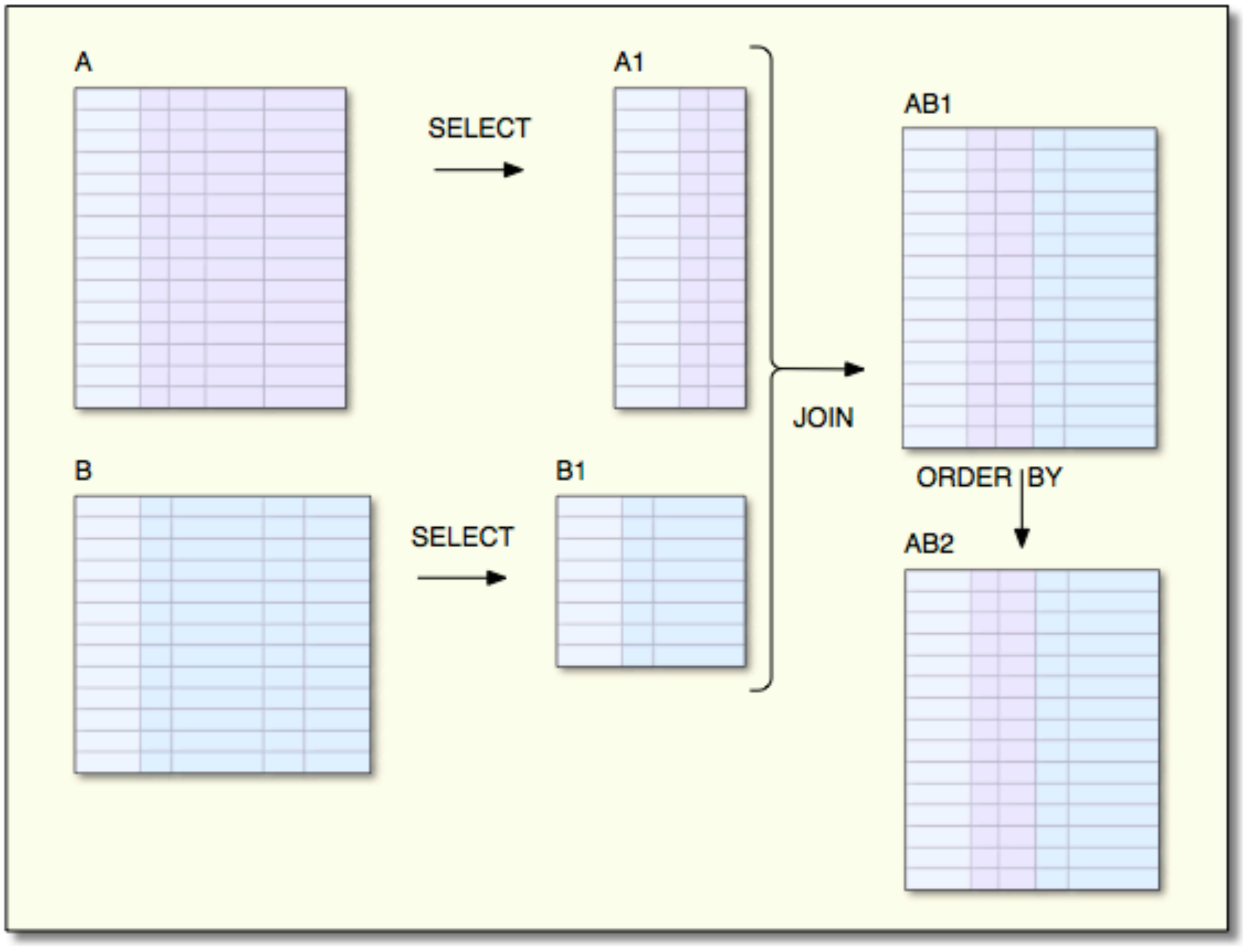


Figure 8. The Interweaving of Threads

Customer table and **B** might be an *Order* table and they have a *Customer_ID* in common. We represent the common element as the white column in the table.

The diagram represents two **SELECT**s being carried out on the two tables, **A** and **B**. The results of the **SELECT**s are **JOIN**ed and the result of the **JOIN** is sorted into a specific **ORDER**. So if we represent the **SELECT** function as *S1* and *S2*, the **JOIN** function as ***J*** and the **ORDER BY** function as ***O***, we can represent this diagram algebraically by writing:

AB2 = ***O***(**AB1**), **AB1** = ***J***(**A1**, **A2**), **A1** = ***S1***(**A1**), **B1** = ***S2***(**B**)

But did we have to do it in that order? Maybe we didn't. Maybe, for example, we could have done the **JOIN** first and applied the **SELECT**s afterwards. That kind of calculation, by the way, is what RDBMS optimizers tend to work out - when to do **JOIN**s, which **JOIN**s to do first and so on. So these are options that A^{2}DB may ponder as well and it will make a choice. Assuming it chooses the solution: **AB2** = ***O***(**AB1**), **AB1** = ***J***(**A1**, **A2**), **A1** = ***S1***(**A1**), **B1** = ***S2***(**B**), it will work out all those sets and the **Set Processor** will return the answer **AB2**.

The **Set Processor** then passes these **clans** to the **Storage Manager**, and the **Storage Manager** does the following:

- It transforms the **clans A1**, **B1**, **AB1** and **AB2** into a **PHYSICAL** form appropriate for storage.

- It tells the **Resource Manager** what these **clans** are, where they are and how it stored them.
- It tells the **Universe Manager** that the universe of **clans** includes four more **clans**, what they are and how they are related algebraically.

So now we can explain why we previously wrote *"This will yield one or more possible solutions."*

A²DB retains all its intermediate results for reuse. It keeps on doing this until it starts to run short of space. When it runs short of space, it discards intermediate results that have not proved useful. When a query arrives, A²DB may hold many different **clans** that could contribute to providing an answer to the query. In fact, in some cases it may already hold the answer. A²DB has many **LOGICAL clans** to choose from. A²DB does the math, choosing the most promising solutions and eliminating others. Then it does the math again to provide the fastest **PHYSICAL** solution.

The Capabilities and Performance of A²DB

It is important to note that, at the time of writing, A²DB is in its first release and has only been deployed in a number of beta sites and in various proof-of-concept projects. There are two areas of deployment where it is too early to comment on A²DB's capability:

1. **In OLAP application areas.** Because of its self tuning capability, A²DB is capable of providing an OLAP service if user query patterns resemble those that would be typical of OLAP usage. However, it doesn't currently provide an MDX (Multidimensional Expressions) interface. Once OLAP capability is provided (as is intended) it will be possible to comment on this kind of usage.
2. **In large scale-out applications.** A²DB is currently only written for Symmetric Multiprocessing (SMP) deployment. As with the OLTP capability, a Massively Parallel Processing (MPP) capability is intended for the next release. There is indication that there will not be any problem in adjusting the architecture for MPP operation.

Setting these application areas aside, there are good reasons to believe that A²DB will equal or outperform competitor products in most situations, for the following reasons:

- A**2**DB applies the same data placement techniques that other databases use to retrieve data quickly.
- Because of the way it works, A**2**DB speeds up. The more it is used, the faster it gets.

This deserves some explanation. As we previously noted, there are only a limited number of techniques for speeding up data retrieval; data partitioning, data compression, efficient indexing, efficient parallel processing and hardware acceleration (using purpose built processors). Other performance-related terminology like "column store" or "statistical optimization" or "bit-mapped indexing" are simply other terms that refer to nuances of one or more of these techniques. A²DB currently deploys many of these techniques, in situations where it calculates that they will be effective.

A BI database's workload consists of a queue of queries that require answers. The queries are usually processed one-at-a-time, so that the activity in retrieving data for one query does not interfere with retrieving data for another. Memory is much faster than disk (by a factor of 10,000 to 100,000), so databases try to ensure that the data needed for a query is held in memory if at all possible. In almost all circumstances it is best to provide the BI database server(s) with as much memory as possible. Similarly all BI databases have parallel architectures and attempt to make effective use of CPU cores, by trying to keep them all busy on a query until the answer has been assembled.

Relational databases used to make extensive use of indexes, storing data in btree file structures, with DBAs adding new indexes or removing them in an effort to “tune” the database. But this technique has ceased to be very effective. For this reason a “new generation” of column store databases, the first of which was Sybase IQ, emerged. Such databases are most effective when a specific table (often called the Fact Table) within the BI database is very large. They partition the table in columns and spread the columns over different disks or even different servers. This allows them to parallelize data retrieval in a very effective way. An additional advantage of this approach is that the need for DBAs is reduced.

A²DB makes the same use of “column store” techniques as other BI databases. In fact, at the physical level, A²DB can deploy any software-based optimization technique known to man because it can represent any approach to data storage algebraically. It can deploy all the performance weapons that other products do. However, A²DB differs from other products in two important ways:

1. It self-tunes
2. It retains results and intermediate sets

If the absolute best solution is, for example, to split up tables into columns and arrange those columns in a particular way on disk, A²DB will start to migrate its data in the direction of that arrangement. This will happen naturally in the way that the product works. If column storage is just what's needed, this will become apparent in the queries that arrive, because A²DB will discover that columns start to occur frequently in its intermediate results and it will store those columns.

However, because it stores all intermediate results, it will also store subsets of columns or tables, which will likely be even more useful (i.e. faster to process) in providing answers to user requests. This is why it is difficult to assess the raw performance of A²DB in comparison with other products.

If you use a standard benchmark, A²DB will simply read the answer, probably from memory, the second time you run the benchmark. It will be unbeatable.

Performance and the Brevity of Answers

It's a simple fact that most database queries have short answers. It's easy to provide examples, such as "show me our top 1000 customers along with the average number of orders they place each quarter" or “show me the 100 products that were most popular over the holiday season.” Users cannot meaningfully deduce much from huge volumes of data, so they don’t ask for them unless they are trying to build a database subset to analyze in isolation. However, with A²DB there's no need to do that kind of extract. They can query the database as is.

The fact that most requests for data are actually for small amounts of data works to A²DB's considerable advantage. That, coupled with the fact that users tend to ask the same types of question, with some variance but not a great deal, means that it's very likely that A²DB will already have intermediate results ready that will help it produce an answer to new queries very quickly - faster than competitive products.

Recording intermediate information naturally has an overhead. In practice this turns out to be somewhere in the region of 6 to 10 times the data volume of the initial CSV files that are loaded into A²DB in typical applications. This is the disk space that A²DB will tend to take if it is not constrained and forced to use less. This is a resource trade-off that A²DB makes. Disk is cheaper than memory, so it is better to make maximum use of disk. However A²DB will happily work with less disk space. When doing so, it simply discards some of the intermediate results it might otherwise hold. The performance of A²DB will be compromised to some degree when this is done, and there is little point in imposing such a constraint.

Point-In-Time Queries: A Unique Capability

A known but rarely discussed problem with databases involves updating data. For decades databases allowed updates to data as a standard capability. Unfortunately data updates destroy information. If you replace one value for an item with another you no longer have any record of what the old value was. For example, if a customer credit limit of $10,000 gets changed to $15,000, the fact that it was once $10,000 is lost. Updates destroy data.

This is a larger problem than it might seem on the surface. Consider the situation where, in April say, you run a report to list and add up the credit limits for all customers. It will report the amount of credit that you are making available to customers in total. Having looked at the report, you now want to know, for the sake of comparison, how much credit you were making available to customers at the beginning of January. You have no way of finding out.

You can fix this problem, by archiving all changes and taking snapshots of tables, but it is a cumbersome solution. It is far better if the database allows neither direct updates nor deletes. A simple solution is to represent deletions of records by a flag and date/time when the record was registered as deleted. It is then simple to register an update as a deletion with date/time, followed by a new record with date/time which is flagged as a change to information.

This is how A²DB handles updates, and as far as we're aware, it is unique in doing so. This is a natural side-effect of how A²DB works. When new records are added to a clan, A²DB represents this algebraically as a **union** between one clan and another. So an update has to be represented as a delete followed by an insertion *so that the union operation makes mathematical sense.*

A consequence of this approach is that A²DB is uniquely suited to BI applications that deal with regulation and governance. Because of its approach to data update, A²DB naturally includes a complete audit trail of all changes to data. A single query against an A²DB database will provide a complete audit trail of changes. Additionally it is possible to do point-in-time queries on any set of data. For example: List all staff and their annual salaries as of March 31st 2009 or provide company salary totals for March 31st and June 31st.

The important point to note here is not so much the point-in-time capability as the fact that no additional effort is required in order to gather answers to such queries. With other database products it is possible to finagle a way of getting the information, by doing some specific programming or pulling information from log files. With A²DB no additional effort is required. It just works that way.

The Business Perspective

With most high-performance database products, the motivation for adopting them is that they will deliver better query performance than other alternatives. The typical scenario is that BI activities are being slowed down by the current database in use and an alternative product will speed up the response allowing critical business strategies or decisions to be implemented faster. It may be that there are differences in license costs that are attractive, but generally the product is chosen for BI performance.

In recent years, newer high-performance products claiming to be one or two orders of magnitude faster than traditional database products have emerged. In some contexts these products are indeed faster, mainly because they use a “column store” approach enabling them to handle queries on very large tables in a parallel manner. They also implement on inexpensive commodity x86 servers, so hardware costs can be relatively low. Some also claim not to need any performance tuning by DataBase Administrators (DBAs). Some databases of this ilk are calling themselves ***3rd generation databases.***

By those three criteria, A^{2}DB is a 3rd generation database. It will physically implement column stores, if that it is the optimal way to handle the query traffic; it runs on commodity hardware and it has no DBA requirement for performance tuning. Additionally it can deliver performance that is one or two orders of magnitude faster than a typical relational database.

Beyond the Third Generation

Having noted these similarities with “column store” products, it is important we emphasize that there are very significant differences between A^{2}DB and such products. They are as follows:

A^{2}DB is self-tuning: The reason that “column store” products require little DBA activity is because they focus on partitioning large tables of data over multiple disks and multiple servers. The speed of the database in answering queries depends primarily on how well it does that - in essence how well it spreads data out so that the typical query can be resolved in a parallel manner. Such databases will attempt to maintain an intelligent cache of data in memory so that the need to access disk is minimized.

The reason that such column stores require little DBA activity is twofold.

1. As new data is added to the large tables, the same approach of column partitioning is applied.
2. Most relational databases use indexes to get at data. DBAs that tune such databases for performance spend a great deal of time adding, removing and tuning indexes. The column store databases do not require such activity.

However, to our knowledge, none of the “column store” databases are self-tuning. They apply a specific strategy to performance and where it works well with respect to query traffic, it results in good performance. By contrast, A^{2}DB is self-tuning. The performance is good no matter what the query traffic. And the more it is used the faster it gets.

Currently, its primary limitation is that the MPP capability has not yet been completed. Because of that, A^{2}DB at the moment is best suited for BI databases up to the 1 or 2 terabyte level, which, fortunately for Algebraix Data, includes the vast majority of BI databases. A^{2}DB would need the parallelism that can be achieved with multiple servers in order to cater to databases at the petabyte level.

A²DB requires almost no design effort: With all other database products there is a need to design the data in accordance with the way that the database works. This is typified by the design of star schemas or snowflake schemas for BI databases, so that the performance of the database is not hindered by repeatedly carrying out multi-way joins. Even with OLTP databases, design work usually involves normalizing the data to 3rd normal form and then making design compromises to suit the database product that is being used.

With A²DB, any initial logical design will do. In practice, the most sensible design to begin with is likely to be one that matches the CSV files being extracted from source databases and files. As long as the metadata is properly defined, that approach will work as well as anything else.

A²DB is a true data warehouse: On many occasions companies have built data warehouses that are not used for direct queries, but simply used as a staging area for creating data marts. Data marts are analysis databases - subsets of data drawn from the warehouse for use by a specific group of users. Some data marts are OLAP databases. Sometimes, yet other BI databases, called operational data stores, are built which do not use the data warehouse at all, but draw their data directly from on-line systems. These BI databases are used to circumvent the reality that it can take too long for data to be loaded into the data warehouse. If you want up-to-minute queries, then you need a special data store that is updated quickly. Finally there are streaming databases. These are databases that manage streams of data that are being created in real-time, often including data drawn from OLTP transactions as they are being applied to other systems. These are BI databases of a kind, too, operating in real-time.

Currently A²DB is capable of being a true data warehouse. There is no need to create data subsets, since all queries can be applied to A²DB without fear of the performance issues that arise with other database products. In its current release it is not yet capable of providing full OLAP speed, but that functionality is currently being built into the product. Similarly it is not capable of being an operational data store or a streaming database. Again these are capabilities that are scheduled to be introduced.

A²DB will be a strategic data resource: A primary area of BI activity is Master Data Management (MDM). The software tools that deliver MDM try to ensure that an organization does not use multiple (and potentially inconsistent) versions of the same master data in different applications. A²DB makes no contribution to MDM yet. However, because it has a precise and unambiguous algebraic view of data, it has the potential to become the fundamental MDM resource for an organization.

The A²DB Benefits in Summary

We can view the potential benefits that A²DB confers from two perspectives; reducing costs and creating opportunity. In terms of cost reduction, it delivers the following benefits:

- **Low hardware costs.** It runs on commodity x86 hardware and the executable has a very economic software footprint. Some relatively small BI applications that might otherwise require an MPP configuration, will fit into SMP with A²DB.

- **Scalability:** It scales well (roughly linearly) within SMP environments and, in such environments, will accommodate data growth without performance issues arising. Note however, that it does not yet have MPP capability.
- **Extremely low operational overhead:** A²DB is self-managing in many ways that other databases are not. There is no need to manage resource space allocated. Data take-on and update is simple. Fewer database instances are required, both for physical reasons and for design reasons. In particular there is no requirement for DBAs to tune the database for performance as it is self-tuning.
- **Database design effort reduces almost to zero:** A²DB is extremely forgiving with respect to database design. Any design that is consistent with respect to data is a viable starting point. It is highly unlikely, as sometimes happens, that a project will fail because of design errors. This means that project delays are less likely and less experienced staff can be used for design work.

A²DB supports or creates business opportunity in the following ways:

- **Improved performance and accelerating performance:** A²DB delivers high performance and uniquely among database products, the performance is likely to increase with usage. It is particularly suited to BI applications in the 1- 2 terabyte arena where performance has been an issue for other database products.
- **Faster time to value:** Algebraix Data claims A²DB can deliver a faster time to value than competitive products. This stems from several attributes of the product, including fast load times and the fact that a single instance can be used both as a data warehouse and a collection of data marts. Algebraix Data also claims, with some justification, that it has a wider area of application in BI (as indicated in the following bullet points).
- **Capable of handling dynamic metadata:** Because of A²DB's algebraic management of data, it is better able to handle dynamic metadata, which causes difficulty in some applications, such a clinical trials systems in pharmaceutical companies and derivative trading systems in the banking sector.
- **Particularly suited to sparse data:** Again, because of A²DB's algebraic management of data, it is better able to handle databases that contain large amounts of sparse data.
- **Particularly suited to regulatory data:** A²DB holds a full audit trail of all changes and enables point-in-time queries. This makes it particularly suited to databases that deal with regulatory data or with data assembled from log files of any kind. Web log data is a particular area of application.
- **Strategic foundation:** A²DB will work on some BI applications that have previously proved to be too difficult to implement. Additionally, Algebraix Data claims that, because A²DB is so versatile, it occupies a foundational position within any BI architecture. It reduces ETL effort, eliminates DBA work, manages metadata changes. In general, BI applications that were previously impossible become possible.

About The Bloor Group