AmorphousDB: A New Take on an Old Problem

Introduction

Relational databases had a good run for almost fifty years, but they are no longer capable of managing modern data. Anyone can search the entire Internet in seconds, asking for data that a trained database expert would need hours to find in a single corporate database.

When commercial relational database systems appeared in the late 1970s, they were a huge improvement on the earlier database systems and a natural fit for departmental computing on the increasingly popular 32-bit minicomputers. A little later, with a national standard built around the SQL language, relational database systems achieved effective hegemony in both commercial computing and academic interests. SQL database systems were based on a bastardized version of Codd's Relational Data Model and kept trappings from relational algebra on which it was based.

But looking at relational database systems from the perspective of search engine technology (in which retrieval must merge indexes from various words, synonyms, corrected spelling, singularizations, etc.), shows that the relational model carries a great deal of excess baggage yet fails to address common needs

For example, dropping the concept of table and giving records a record type name does not change anything in SQL syntax or semantics. However, eliminating tables also eliminates the requirement that all records of given type have the same attributes creating the ability to add attributes to existing records as the database evolves or the requirements change, bypassing the much-hated database schema. Substituting a dedicated data type that links primary and foreign keys retains the "link by common value" that made relational databases so successful and avoids the various traps inherent in declared keys.

Neither of these changes has any effect on SQL syntax or semantics. Shedding the relational dogma that all retrievals of related records must be returned as tables and adding language that returns related records as sub-lists adds a powerful and application-friendly capability. This change, unfortunately, does break the SQL language. Keeping SQL but adding a more powerful language makes sense.

These observations begat the Amorphous Data Model (meaning, no data model), the Amorphous Data Access Language (ADL), and AmorphousDB, an implementation. The Amorphous project began with the Relational Database, shed unnecessary baggage and restrictions, added a data type to simplify managing record relationships, cleaned up and rationalized the API, and invented a language to provide access to the newfound flexibility. While simpler and more powerful that relational database systems, AmorphousDB retains the ability to model relational databases and execute SQL.

A second and equally ambitious goal of Amorphous, also inspired by search engines, is to develop a highly scalable, multi-server, multi-site implementation architecture that supports ACID transactions and sends operations to the data rather than dragging the data to the operation.

Why Amorphous? What are the Problems?

Problem #1: A seventy year old geezer can search the untold billions of web pages with Google in a second or so. A trained and experienced database expert would take days to do an equivalent search against a corporate relational database and would require specialized software to read and analyze every record into the database to do so.

Problem #2: If Amazon were to use a relational database for product data, a typical product page (for example a Samsung SDD) would require 81 records from 17 tables to generate the page. The SQL query to fetch the required data, would return 419,904,000 rows. (Amazon, unsurprisingly, doesn't use a relational database for product data).

Problem #3: Most relational database systems don't scale beyond a single server. Those that do often require moving gigabytes around a network to resolve a simple query.

Problem #3: Relational database systems deal with character and numeric data. But much of the world's information is in documents – PDFs, Microsoft Word and PowerPoint, etc. Most relational database systems can store and retrieve but not search documents.

Problem #5: Modern development technology depends on evolution and agile response to changes in application requirements. Changing the schema of a relational database often requires off-line restructuring, which is incompatible with availability and far from agile.

The Amorphous Solutions

Solution #1: Amorphous merges search engine technology with structured data while preserving ACID transactions. To complement this, high level language procedures in Amorphous provide browser access to a database without code or an application server.

Solution #2: Amorphous executes sub-queries in parallel and returns related records in sub-lists rather than a massive table. This is faster, more efficient, and more application friendly than relational joins.

Solution #3: An Amorphous database can span many sites and servers for availability, scalable throughput, an optimize local access. Rather than bring data to an operation, Amorphous sends the operation to the data and aggregates the results.

Solution #4: Amorphous has no metadata operations that require any kind of restructuring. Amorphous can add new record types and new attributes for existing records at any time.

Structured versus Flat Retrievals

Most relational applications deal with record relationships, mostly one to many. A typical Amazon product page, for example, will have photos, sizes, styles, colors, service options, protection plans, other sellers, bought together, sponsored products, etc. One way a relational database can retrieve the product and all related records is with a massive outer join producing a table with an infeasible number of rows. Another solution is a series of sequential retrievals which incurs a heavy latency penalty. A third is to issue concurrent retrievals on many database connections, forcing otherwise simple apps to become multi-threaded, breaking transaction atomicity, and increasing system overhead. In each of these cases, the app developer spends his time fighting the database rather than simply exploiting it.

Amorphous and ADL returns data from related records as named sub-lists along with the attributes of the primary record, reducing application complexity and development costs and preserving transaction atomicity. But this isn't simply an access language issue. Relational APIs and network plumbing rely on the principle that all rows returned have the same set of columns. Amorphous, however, returns data in a self-describing structure with no assumption of row uniformity.

Of Nomenclature and History

In the beginning, there was the punched card. Adjoining columns were grouped in "fields", e.g., columns 1 through 20 might be last name, columns 21 to 30 might be first name, etc.

When tape drives and disks were invented card images were stored in" files" and generalized into "records", shedding the 80-character length and acquiring the ability to store binary data and repeating groups of fields, but retaining the concept of fixed length fields. Moreover, the format of the record belonged to the programmer and not the input/output system.

Database systems were invented to abstract record fields into data items of known type, manage record storage on persistent media, and manage relationships among various records. Examples of early database systems include IBM's IMS, which managed records in a hierarchy and the so-called CODASYL network model that linked related records into chains.

Dr. E. F. Codd proposed the Relational Data Model in 1970 based on set-theoretic relational algebra. In his model, "relations" were defined as sets of "n-tuples" of "data items" that could be manipulated by the restrict, join, and project operators to define other relations. In his papers, Dr. Codd used a language notation called Alpha to manage and manipulate relations, tuples, and data items. Since relations were based on set theory where all elements had to be unique, Dr. Codd's operators automatically eliminated duplicate tuples. Consequently, to project a relation of first grade students to a relation containing only first name to print name tags, only one "Emma" name tag would print without regard to the actual number of Emmas in the class. Virtually all database system developers saw automatic duplicate elimination as too error prone to implement, bastardizing Codd's Relational Model.

Dr. Michael Stonebreaker of UC Berkeley developed a database system based on Codd's work called Ingres with an Alpha-based access language call Quel. Human factor studies show that Quel can be taught to non-programmers with great difficulty but simply cannot be taught to programmers who were trained to think algorithmically.

Researchers at IBM developed an alternative database access language, Sequel, originally a pun on Quel but morphed by IBM marketing to SQL, a retronym for Structured Query Language. In the process, they turned Codd's relations into tables, n-tuples into rows, and data items into columns.

So, historically, we have three nomenclatures: file/record/field, relation/tuple/data item, and table/row/column all three describing the same elements.

The AmorphousDB database is based on the Amorphous Data Model, which is to say, no data model at all. Amorphous uses the terms "value" for a single datum of specific type, "attribute" as a named data type, and "record" as an arbitrary collection of attribute/value pairs.

Search

SQL supports data selection more than actual search where users must know exactly what they are looking for and where to find it. For example, the following two equivalent queries would find data about sailboats manufactured by “Hinckley”.

ADL: fetch * from sailboats with manufacturer == “Hinckley”

SQL: select * from sailboats with manufacturer = “Hinckley”

Neither of these, however, would find any sailboat with manufacturer styled as “HINCKLEY” (selection is case sensitive) or “The Hinckley Company” (equality must match the full string. This could be addressed by the following:

ADL: fetch * from sailboats with manufacturer containing “Hinckley”

SQL: No equivalent

But this does require that the user know whether “Hinckley” is the manufacturer, the brand, or possible the designer. The following query gets around this problem:

ADL: fetch * from sailboats with “Hinckley”

SQL: No equivalent

And if all else failed, there is the query:

ADL: fetch * with “Hinckley”

SQL: No equivalent

This, of course, is where a web search engine would start.

But the problems of search in relational databases go far beyond language, case sensitivity, and other operator issues. Relational databases APIs and network plumbing (ODBC, JDBC, proprietary product interfaces) are based on the relational dogma that queries return only tables of rows with like columns. Using self-describing data structures Amorphous returns only the required data in a format easily managed by programs.

Anatomy of an Amorphous Database

While the roots of Amorphous are in relational databases, it is not a relational database. Amorphous is what you have left after dropping the unnecessary, unwieldy, inefficient features of the relational database and adding developer friendly features. It has dataspaces but no schemas. It has records but no tables. It returns self-describing hierarchical data structures, not one wide long table.

An Amorphous database consists of two or more “dataspaces.” Dataspaces share database access accounts but are otherwise independent. Every Amorphous database has a system dataspace used by

the database itself and one or more user dataspaces. Each dataspace has its own metadata, records, and security policies.

Amorphous supports a wide range of data types, including traditional database types: character string, Date, Timestamp, Latitude/Longitude, Boolean, UUID, and Array. Others are Amorphous specific:

- Text, parsed and indexed as individual words, numbers, dates, URLs, email addresses, etc.
- Number, which may be either a scaled integer, a double precision number, or an arbitrary precision integer
- Opaque, like a Blob but with a set of attributes including mime type
- Enumeration, restricted to specific values
- Mark, a dedicated type to establish inter-record relationships.

An “attribute” is a named type of value within a dataspace. Attributes are strongly typed: Any assignment to an attribute is converted to that type before indexing and storage. An application developer can declare attributes explicitly or implicitly within a declare record constraint, or declare them implicitly by a first assignment - whether this is a promising idea or not is TBD. Attributes can be declared as indexed by default, as “metadata”, which requires an explicit attribute name for retrieval, or non-indexed Character string values attributes can be declared as either case sensitive or case insensitive.

A “record” is a persistent set of attribute/value pairs created by the ADL (Amorphous Data Access Language) or SQL “insert” statements. Amorphous does not organize records into tables and there is no requirement that any two records have the same set of attributes. A record can have an optional “record type” implemented with a system defined, text valued, metadata attribute RECORD_TYPE. ADL uses record type names the way SQL database systems use table names.

An application developer can constrain record types with “record constraints” that define validity rules for a named record type. A record constraint can declare a minimum set of required attributes, limit the set of optional attributes, or allow any other attributes for the named record type.

Amorphous includes triggers declared for specific record types to perform actions before or after update operations.

Amorphous can create named mappings to map external keys or data values into internal marks. The relational database import facility, for examples, uses mapping to map primary and foreign keys into marks.

Key Differences between Amorphous and SQL Database Systems

In the relational data model, records exist only in tables and all records with a given table have the same columns. Amorphous does not have tables but allows records to have mutable record types. Unless constrained by a declared record constraint, an Amorphous record can have variable sets of attributes. In addition, subject again to declared record constraints, Amorphous can add additional attribute/value pairs in existing records.

In SQL database systems, record relationships are based on declared primary and foreign keys. Amorphous maintains record relationships with a dedicated data type, the mark. A mark uniquely identifies a record. Records reference each other using their marks.

SQL database systems have explicit numeric types for decimal, numeric, small int, int, big int, float, and double precision. Amorphous has a single abstract numeric type.

In SQL database systems, character-based data is either fixed length or have a declare maximum length. In Amorphous, neither character strings nor text are bounded in length.

In SQL database systems, character-based data is a simple string. In Amorphous, text data is indexed within a string as a sequence of words, numbers, dates, etc. Amorphous has a separate type, character string, which it treats as a single string.

In SQL based database systems, retrieval is table and column based and performed on either a full case sensitive character string or the leading characters of a character string. In Amorphous selective retrievals, the record type is optional. Amorphous selects records by word, phrase, or exact match, in named attributes, or any record attribute, depending on search operator.

In SQL based database systems, character strings are case sensitive, although built-in functions may coerce them to upper or lower case with various effects on indexes. In Amorphous, text is case insensitive. Character strings declarations include whether the attribute is case sensitive or case insensitive.

Amorphous API

Amorphous has simple object-oriented API. A Connection object manages a connection to an Amorphous database. The Amorphous data language supports complex requests with branching, looping, exception handling, and other programming language features. Requests typically include a significant programming step, not just a single select statement. From a connection and an access language statement, an application creates a Request object. Amorphous appends data required by a request instance to the Request object as one or more rows of named self-describing data items. Successful running of the Request object returns a self-describing Result data structure. A failure throws an exception. By default, running a request starts a new transaction. A successful request commits and a failed request rolls back the transaction. A program can choose to manage its own transactions if that is appropriate. SQL applications suffer from the pervasive application bug of forgetting to check the sqlcode after a database operation and thus passing over failed requests and the garbage data they produce.

The Amorphous API is pure virtual interfaces in C++ and interfaces Java. Drivers for other application languages can layer one of these or run natively using the Amorphous client protocol.

Amorphous Data Access Language (ADL)

The Amorphous Data Access Language enables an application to package the input data with the business logic in single request, execute the request, and return all results to the application in a single round trip to the database server.

SQL is a command structured language where the database engine performs a single select, insert, update, or delete. The client application must perform all application logic flow. In contrast, the Amorphous Data Access Language is block structured. Database actions are combined with looping, conditional, and exception handling constructs to enable a single request to encapsulate the full business logic for processing stage and execute it in a single round trip between client and server.

Security Model

Security and access control is both more important and more complex than when they were when SQL was standardized 40 years ago. Modern application must control differential access to database administrators and within applications, administrators, moderators, vetted site members, and unknown members of the public.

In the SQL security, a user logs onto a database, possibly including a pre-defined role. The database gives access rights based on login account and role, but the role cannot change during a session. That is fine if your model is Students and Classes, and students are allowed only to see but not change their own grades. Suppose there is a desire to let students know how they are performing compared to the rest of the class. That is easy if the application can change its access for the purpose of the calculation. Virtually all modern database applications require that the application access data that the application client should not, the responsible for security devolves from the database system to the developer.

In Amorphous, a login account can have many distinct roles in various dataspaces. Each assigned role may be active or latent, and the application can activate and deactivate account roles. Dataspace specific security policies accord access rights to active roles following to declared access right policies. Security policies, in turn, attach to records and other database objects. In SQL databases an access violation results in a fatal error. In Amorphous, lack of access permission results in a record being invisible to the application. Update and control security violations throw exceptions that can be handled within a request.

Embedded Applications

An embedded application is an application hosted by the database server, obviating the need to a separate application server. An embedded application has call level access to the database engine for superior performance. Eliminating an application server reduces application latency for a superior user experience and centralizes application logic. It also provides single point database and application administration. Moving logic to the database system simplifies client development. If the application is web based, it can eliminate the client altogether in favor of a web browser.

But to successfully embed an application, a database system must:

- Run application code in a security sandbox
- Provide multi-node scalability so a single database server never gets bogged down in application code
- Enable dynamic application update without perturbing database availability
- Ensure that a running application instance sees a consistent view of the application while the application itself is upgraded.

- Provide an environment that enables external application development in standard tools before an application is ready to deploy.

Amorphous utilizes a tightly integrated Java Virtual Machine and a custom Java class loader for access Java class definitions store in the system dataspace. The Java subsystem is transaction aware so that a class loader in conjunctions with Amorphous's multi-version concurrent control (MVCC) so a running transaction sees a consistent application version.

Proof of Concept Application

The Amorphous Proof of Concept Application (POCA) is a Web facing embedded application based on the full application inheritance concept. POCA, fully implemented, is a hierarchy of modules, each in its own dataspace. It inherits and can extend and override semantics (Java classes), appearance (HTML templates, images, and data model (but not data) through an inheritance hierarchy.

The foundation of POCA is the "Base application" which has basic support for

- Simple database operation such as search, query, report, and data entry
- Database navigation
- Relational database import (via JDBC)
- Application/module creation, installation, update, backup, and restore,
- Security model management

A fully developed POCA application has four tiers: The base application, a set of pre-defined modules like Members, Calendar, News, and Content, the application itself, and one or more application instances, each with its own data and appearance.

Database Events

Modern applications require real time notification of changes to a shared database by other users. Rather than requiring application developers to build their own notification subsystem (and, frankly, because Amorphous required one itself), Amorphous provides a comprehensive event notification system. Triggers or ADL requests can post events that carry self-describing data and are transactional so clients never receive an event notification before the posting transaction commits. Events have multi-segment names so a client can register an event listener containing wild cards. For security, event names can be pre-declared and associated with a security token to regulate who can post and who can receive specific notifications.

Implementation

Amorphous implements the "atom programming model" in which the database engine is layered on persistent distributed objects called atoms. A fully configured physical Amorphous database consists of two or more sites. Each site consists of two or more servers – computers to the lay person. Each site contains a full set of atoms, but individual servers within a site will host only a subset of atoms. However, each server maintains a complete catalog of which servers contain which atoms.

Each atom in a database replicates on a peer to peer basis with all other instances on itself, within a site and across sites. For durability, Amorphous writes Atoms to local storage and drops them from memory when they are not active. Each atom has a single server, site-wide, that acts as the "chairman" that handles operations that must be serialized for consistency. If a server hosting one or more atom chairmen fails, each atom has a deterministic algorithm to identify a successor chairman. There are no "master" servers and no single points of failure (assuming multiple servers, of course).

When a server requires access to an atom that is not present locally, the situation is managed in one of two ways, depending on the assignment of atoms. If the atomic assignment heuristic says that that server should host that atom, the server will request a copy from the most responsive server containing a copy. If, on the other, the assignment heuristic determines that server should not host that atom, the server will package the request to the most responsive server than has the atom.

Geo-Distribution

Amorphous is dynamically scalable, multi-node, multi-site distributed database. An Amorphous database can be deployed on from one to several hundred database servers. Individual servers are assigned to a site, each of which hosts a full copy of the database. Within a site, database atoms are distributed to declared number of sectors and, if the site is fully configured, each server in the sites hosts atoms within a single sector. A site also has a declared redundancy factor, which is the minimum number of copies on an atom within the site. If a site is less than fully configured (less than the number of sectors times the redundancy factor), site servers in different sectors will host additional atoms to meet the redundancy factor, so adding or removing servers from a site or changing the number of sectors or redundancy factors will kick off a local re-organization.

Connection Brokers and Protocols

A client supplied database connection string references a connection broker rather than a database server. Based on locality, any special client requirements (access to given relational database for import, for sample), and current server loads, the connection broker will assign the client to specific server, to which the client then connects with a crypto handshake. With the sole exception of the initial connection to a connection broker, all connections are first authenticated by the Secure Password Protocol (SRP) and then encrypted with a single use session key, AES, cypher block chaining, and a value based binary encoding.

Appendix

James A. Starkey

AmorphousDB's Chief Cook and Bottle Washer

Starkey joined the ARPAnet Datacomputer project at the Computer Corporation of America in 1973 following his graduation from the University of Wisconsin, Madison (BA in mathematics with Highest Honors). While at CCA he developed the final iteration of Datalanguage, was exposed to early drafts of Codd's relational database papers and concluded that relational database systems where the wave of the future. In any case, Starkey found his place at the cutting edge where the network met the disk.

In 1975 Starkey joined the Digital Equipment Corporation to write a relational database system (the bait) but was assigned to the PDP-11 port of the IDMS mainframe database system (the switch). The resulting product, DBMS-11, worked but was best described as an elephant riding a bicycle. Following DBMS-11 Starkey designed, implemented, and shipped Datatrieve-11, the first of the wildly successful Datatrieve products. In the process, Starkey apparently started the software product mascot tradition when a customer found and exposed the "help wombat" Easter egg. Datatrieve-11 and the FORTRAN compiler were the first two layered products on VAX/VMS.

Shortly after the launch of the VAX 780 Starkey began the development of VAX Datatrieve, which he led until it shipped in 1981. VAX Datatrieve was the core of DEC's VAX Information Architecture. VAX Datatrieve pushed the state of the art in several dimensions, the very least of which was a "plot wombat" to augment "help wombat". In the same timeframe, Starkey started and spun off several other projects, including the Common Data Dictionary and an "official" relational database project. Datatrieve, incidentally, outlived both the VAX and DEC itself, last seen on the market as HP Datatrieve.

Following completion of VAX Datatrieve 1.0, Starkey started on experimental relation database based on his recently invented Multi-Version Concurrency Control (MVCC is now a mainstay of the database industry). DEC's Storage Engineer picked the experimental database (JRD – Jim's Relational Database) for the core of future database machine product. DEC adopted JRD's interface architecture as the DEC Standard Relational Interface and JRD was productized as the first of the Rdb products on both VMS and Dave Cutler's VAX/Eln operating systems.

In 1984, Starkey left DEC to create what became Interbase Software Corporation, purveyor of fine relational database software to the aerospace and semi-conductor industries. Interbase was self-funded, depending on partnerships and borrowed machines for the first year or two. Interbase entered into a staged acquisition with Ashton-Tate that was completed in 1991, shortly before Borland bought Ashton-Tate for the Interbase technology. Starkey left the company after the Ashton-Tate acquisition. Interbase continues to this day as a product from Embarcadero and the open-source database Firebird.

Waiting out a non-compete, Starkey started Harbor Software which developed Harborview, an innovative, syntax-free, graphical application development platform. Unfortunately, like virtually all of its competitors, Harbor Software starved to death during the extended period between the announcement and the actual launch of Microsoft Access.

Circa 2000, Starkey started Netfrastructure, Inc., a Web application platform that integrated a page generation engine, a Java virtual machine, and Starkey's third transactional relational database system. MySQL AB acquired Netfrastructure in 2006 to convert the Netfrastructure relational database system into the MySQL Falcon storage engine. During his period with MySQL Starkey developed some seminal ideas for a radically new database architecture that had to be put on ice when Sun Microsystems acquired MySQL.

In 2008 Starkey founded the company that became NuoDB, Inc. to develop a multi-node, multi-site, elastic, distributed relational database system based on the "atom programming model." An omnibus patent sailed through the US Patent Office in under a year after a finding of "no prior art." Starkey retired from NuoDB in 2012.

Retirement didn't "take", however, as ideas are hard to stop. And from the never-ending flow of ideas came AmorphousDB.

Mr. Starkey has received six patents on various aspects of database technology and has twice been invited to speak in the Boston joint ACM/IEEE lecture series.

Jim Starkey lives in Manchester (-by-the-Sea) Massachusetts with his wife and co-conspirator, Ann Harrison, and their dog, two cats, and a sailboat.

Jim can be reached at jim@jimstarkey.net