Database Management Systems

eBooks For All Edition
(www.ebooks-for-all.com)

PDF generated using the open source mwlib toolkit. See http://code.pediapress.com/ for more information.
PDF generated at: Sun, 20 Oct 2013 01:48:50 UTC

Contents

Articles

Database 1
Database model 16
Database normalization 23
Database storage structures 31
Distributed database 33
Federated database system 36
Referential integrity 40
Relational algebra 41
Relational calculus 53
Relational database 53
Relational database management system 57
Relational model 59
Object-relational database 69
Transaction processing 72

Concepts

76
ACID 76
Create, read, update and delete 79
Null (SQL) 80
Candidate key 96
Foreign key 98
Unique key 102
Superkey 105
Surrogate key 107
Armstrong's axioms 111

Objects

113
Relation (database) 113
Table (database) 115
Column (database) 116
Row (database) 117
View (SQL) 118
Database transaction 120
Transaction log 123

Database trigger 124
Database index 130
Stored procedure 135
Cursor (databases) 138
Partition (database) 143

Components **145**
Concurrency control 145
Data dictionary 152
Java Database Connectivity 154
XQuery API for Java 157
ODBC 163
Query language 169
Query optimization 170
Query plan 173

Functions **175**
Database administration and automation 175
Replication (computing) 177

Database Products **183**
Comparison of object database management systems 183
Comparison of object-relational database management systems 185
List of relational database management systems 187
Comparison of relational database management systems 190
Document-oriented database 213
Graph database 217
NoSQL 226
NewSQL 232

References
Article Sources and Contributors 234
Image Sources, Licenses and Contributors 240

Article Licenses
License 241

Database

A **database** is an organized collection of data. The data are typically organized to model relevant aspects of reality in a way that supports processes requiring this information. For example, modeling the availability of rooms in hotels in a way that supports finding a hotel with vacancies.

Database management systems (DBMSs) are specially designed applications that interact with the user, other applications, and the database itself to capture and analyze data. A general-purpose **database management system (DBMS)** is a software system designed to allow the definition, creation, querying, update, and administration of databases. Well-known DBMSs include MySQL, PostgreSQL, SQLite, Microsoft SQL Server,Oracle, SAP, dBASE, FoxPro, IBM DB2, LibreOffice Base and FileMaker Pro. A database is not generally portable across different DBMS, but different DBMSs can by using standards such as SQL and ODBC or JDBC to allow a single application to work with more than one database.

Terminology and overview

Formally, the term "database" refers to the data itself and supporting data structures. Databases are created to operate large quantities of information by inputting, storing, retrieving, and managing that information. Databases are set up so that one set of software programs provides all users with access to all the data.

A "database management system" (DBMS) is a suite of computer software providing the interface between users and a database or databases. Because they are so closely related, the term "database" when used casually often refers to both a DBMS and the data it manipulates.

Outside the world of professional information technology, the term *database* is sometimes used casually to refer to any collection of data (perhaps a spreadsheet, maybe even a card index). This article is concerned only with databases where the size and usage requirements necessitate use of a database management system.[1]

The interactions catered for by most existing DBMS fall into four main groups:

- Data definition. Defining new data structures for a database, removing data structures from the database, modifying the structure of existing data.
- Update. Inserting, modifying, and deleting data.
- Retrieval. Obtaining information either for end-user queries and reports or for processing by applications.
- Administration. Registering and monitoring users, enforcing data security, monitoring performance, maintaining data integrity, dealing with concurrency control, and recovering information if the system fails.

A DBMS is responsible for maintaining the integrity and security of stored data, and for recovering information if the system fails.

Both a database and its DBMS conform to the principles of a particular database model.[2] "Database system" refers collectively to the database model, database management system, and database.[3]

Physically, database servers are dedicated computers that hold the actual databases and run only the DBMS and related software. Database servers are usually multiprocessor computers, with generous memory and RAID disk arrays used for stable storage. RAID is used for recovery of data if any of the disks fails. Hardware database accelerators, connected to one or more servers via a high-speed channel, are also used in large volume transaction processing environments. DBMSs are found at the heart of most database applications. DBMSs may be built around a custom multitasking kernel with built-in networking support, but modern DBMSs typically rely on a standard operating system to provide these functions. [*citation needed*] Since DBMSs comprise a significant economical market, computer and storage vendors often take into account DBMS requirements in their own development plans.[*citation needed*]

Databases and DBMSs can be categorized according to the database model(s) that they support (such as relational or XML), the type(s) of computer they run on (from a server cluster to a mobile phone), the query language(s) used to access the database (such as SQL or XQuery), and their internal engineering, which affects performance, scalability, resilience, and security.

Applications and roles

Most organizations in developed countries today depend on databases for their business operations. Increasingly, databases are not only used to support the internal operations of the organization, but also to underpin its online interactions with customers and suppliers (see Enterprise software). Databases are not used only to hold administrative information, but are often embedded within applications to hold more specialized data: for example engineering data or economic models. Examples of database applications include computerized library systems, flight reservation systems, and computerized parts inventory systems.

Client-server or transactional DBMSs are often complex to maintain high performance, availability and security when many users are querying and updating the database at the same time. Personal, desktop-based database systems tend to be less complex. For example, FileMaker and Microsoft Access come with built-in graphical user interfaces.

General-purpose and special-purpose DBMSs

A DBMS has evolved into a complex software system and its development typically requires thousands of person-years of development effort.[4] Some general-purpose DBMSs such as Adabas, Oracle and DB2 have been undergoing upgrades since the 1970s. General-purpose DBMSs aim to meet the needs of as many applications as possible, which adds to the complexity. However, the fact that their development cost can be spread over a large number of users means that they are often the most cost-effective approach. However, a general-purpose DBMS is not always the optimal solution: in some cases a general-purpose DBMS may introduce unnecessary overhead. Therefore, there are many examples of systems that use special-purpose databases. A common example is an email system: email systems are designed to optimize the handling of email messages, and do not need significant portions of a general-purpose DBMS functionality.

Many databases have application software that accesses the database on behalf of end-users, without exposing the DBMS interface directly. Application programmers may use a wire protocol directly, or more likely through an application programming interface. Database designers and database administrators interact with the DBMS through dedicated interfaces to build and maintain the applications' databases, and thus need some more knowledge and understanding about how DBMSs operate and the DBMSs' external interfaces and tuning parameters.

General-purpose databases are usually developed by one organization or community of programmers, while a different group builds the applications that use it. In many companies, specialized database administrators maintain databases, run reports, and may work on code that runs on the databases themselves (rather than in the client application).

History

With the progress in technology in the areas of processors, computer memory, computer storage and computer networks, the sizes, capabilities, and performance of databases and their respective DBMSs have grown in orders of magnitudes.

The development of database technology can be divided into three eras based on data model or structure: navigational,[5] SQL/relational, and post-relational. The two main early navigational data models were the hierarchical model, epitomized by IBM's IMS system, and the Codasyl model (Network model), implemented in a number of products such as IDMS.

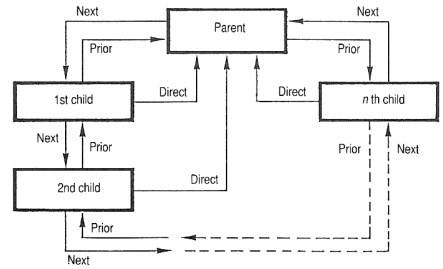
The relational model, first proposed in 1970 by Edgar F. Codd, departed from this tradition by insisting that applications should search for data by content, rather than by following links. The relational model is made up of ledger-style tables, each used for a different type of entity. It was not until the mid-1980s that computing hardware became powerful enough to allow relational systems (DBMSs plus applications) to be widely deployed. By the early 1990s, however, relational systems were dominant for all large-scale data processing applications, and they remain dominant today (2013) except in niche areas. The dominant database language is the standard SQL for the relational model, which has influenced database languages for other data models.[*citation needed*]

Object databases were invented in the 1980s to overcome the inconvenience of object-relational impedance mismatch, which led to the coining of the term "post-relational" but also development of hybrid object-relational databases.

The next generation of post-relational databases in the 2000s became known as NoSQL databases, introducing fast key-value stores and document-oriented databases. A competing "next generation" known as NewSQL databases attempted new implementations that retained the relational/SQL model while aiming to match the high performance of NoSQL compared to commercially available relational DBMSs.

1960s Navigational DBMS

The introduction of the term *database* coincided with the availability of direct-access storage (disks and drums) from the mid-1960s onwards. The term represented a contrast with the tape-based systems of the past, allowing shared interactive use rather than daily batch processing. The Oxford English dictionary cites a 1962 report by the System Development Corporation of California as the first to use the term "data-base" in a specific technical sense.



A closed chain of records in a navigational database model (e.g. CODASYL), with ***next pointers***, ***prior pointers*** and ***direct pointers*** provided by keys in the various records.

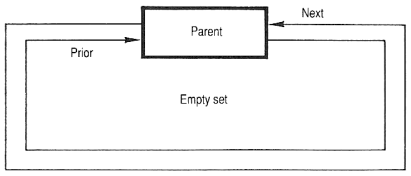


Illustration of an ***empty set***

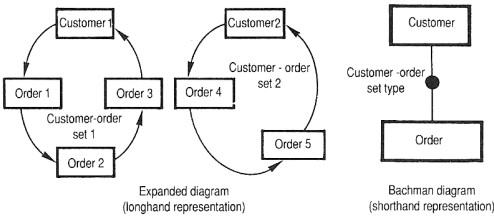


Illustration of a set type using a ***Bachman diagram***

The record set, basic structure of navigational (e.g. CODASYL) databse model. A set consists of one parent record (also called "the owner"), and n child records (also called members records)

Basic structure of navigational CODASYL database model.

As computers grew in speed and capability, a number of general-purpose database systems emerged; by the mid-1960s there were a number of such systems in commercial use. Interest in a standard began to grow, and Charles Bachman, author of one such product, the Integrated Data Store (IDS), founded the "Database Task Group" within CODASYL, the group responsible for the creation and standardization of COBOL. In 1971 they delivered their standard, which generally became known as the "Codasyl approach", and soon a number of commercial products based on this approach were made available.

The Codasyl approach was based on the "manual" navigation of a linked data set which was formed into a large network. Records could be found either by use of a primary key (known as a CALC key, typically implemented by hashing), by navigating relationships (called sets) from one record to another, or by scanning all the records in sequential order. Later systems added B-Trees to provide alternate access paths. Many Codasyl databases also added a query language that was very straightforward. However, in the final tally, CODASYL was very complex and required significant training and effort to produce useful applications.

IBM also had their own DBMS system in 1968, known as *IMS*. IMS was a development of software written for the Apollo program on the System/360. IMS was generally similar in concept to Codasyl, but used a strict hierarchy for

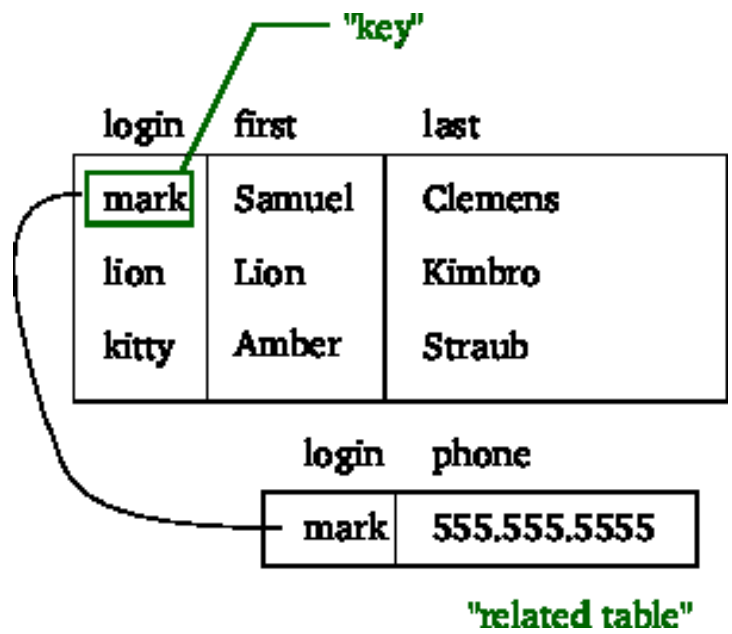
its model of data navigation instead of Codasyl's network model. Both concepts later became known as navigational databases due to the way data was accessed, and Bachman's 1973 Turing Award presentation was *The Programmer as Navigator*. IMS is classified as a hierarchical database. IDMS and Cincom Systems' TOTAL database are classified as network databases.

1970s relational DBMS

Edgar Codd worked at IBM in San Jose, California, in one of their offshoot offices that was primarily involved in the development of hard disk systems. He was unhappy with the navigational model of the Codasyl approach, notably the lack of a "search" facility. In 1970, he wrote a number of papers that outlined a new approach to database construction that eventually culminated in the groundbreaking *A Relational Model of Data for Large Shared Data Banks*.[6]

In this paper, he described a new system for storing and working with large databases. Instead of records being stored in some sort of linked list of free-form records as in Codasyl, Codd's idea was to use a "table" of fixed-length records, with each table used for a different type of entity. A linked-list system would be very inefficient when storing "sparse" databases where some of the data for any one record could be left empty. The relational model solved this by splitting the data into a series of normalized tables (or *relations*), with optional elements being moved out of the main table to where they would take up room only if needed. Data may be freely inserted, deleted and edited in these tables, with the DBMS doing whatever maintenance needed to present a table view to the application/user.

The relational model also allowed the content of the database to evolve without constant rewriting of links and pointers. The relational part comes from entities referencing other entities in what is known as one-to-many relationship, like a traditional hierarchical model, and many-to-many relationship, like a navigational (network) model. Thus, a relational model can express both hierarchical and navigational models, as well as its native tabular model, allowing for pure or combined modeling in terms of these three models, as the application requires.



In the relational model, related records are linked together with a "key"

For instance, a common use of a database system is to track information about users, their name, login information, various addresses and phone numbers. In the navigational approach all of these data would be placed in a single record, and unused items would simply not be placed in the database. In the relational approach, the data would be *normalized* into a user table, an address table and a phone number table (for instance). Records would be created in these optional tables only if the address or phone numbers were actually provided.

Linking the information back together is the key to this system. In the relational model, some bit of information was used as a "key", uniquely defining a particular record. When information was being collected about a user, information stored in the optional tables would be found by searching for this key. For instance, if the login name of a user is unique, addresses and phone numbers for that user would be recorded with the login name as its key. This simple "re-linking" of related data back into a single collection is something that traditional computer languages are

not designed for.

Just as the navigational approach would require programs to loop in order to collect records, the relational approach would require loops to collect information about any *one* record. Codd's solution to the necessary looping was a set-oriented language, a suggestion that would later spawn the ubiquitous SQL. Using a branch of mathematics known as tuple calculus, he demonstrated that such a system could support all the operations of normal databases (inserting, updating etc.) as well as providing a simple system for finding and returning *sets* of data in a single operation.

Codd's paper was picked up by two people at Berkeley, Eugene Wong and Michael Stonebraker. They started a project known as INGRES using funding that had already been allocated for a geographical database project and student programmers to produce code. Beginning in 1973, INGRES delivered its first test products which were generally ready for widespread use in 1979. INGRES was similar to System R in a number of ways, including the use of a "language" for data access, known as QUEL. Over time, INGRES moved to the emerging SQL standard.

IBM itself did one test implementation of the relational model, PRTV, and a production one, Business System 12, both now discontinued. Honeywell wrote MRDS for Multics, and now there are two new implementations: Alphora Dataphor and Rel. Most other DBMS implementations usually called *relational* are actually SQL DBMSs.

In 1970, the University of Michigan began development of the MICRO Information Management System[7] based on D.L. Childs' Set-Theoretic Data model.[8][9][10] Micro was used to manage very large data sets by the US Department of Labor, the U.S. Environmental Protection Agency, and researchers from the University of Alberta, the University of Michigan, and Wayne State University. It ran on IBM mainframe computers using the Michigan Terminal System.[11] The system remained in production until 1998.

Database machines and appliances

In the 1970s and 1980s attempts were made to build database systems with integrated hardware and software. The underlying philosophy was that such integration would provide higher performance at lower cost. Examples were IBM System/38, the early offering of Teradata, and the Britton Lee, Inc. database machine.

Another approach to hardware support for database management was ICL's CAFS accelerator, a hardware disk controller with programmable search capabilities. In the long term, these efforts were generally unsuccessful because specialized database machines could not keep pace with the rapid development and progress of general-purpose computers. Thus most database systems nowadays are software systems running on general-purpose hardware, using general-purpose computer data storage. However this idea is still pursued for certain applications by some companies like Netezza and Oracle (Exadata).

Late-1970s SQL DBMS

IBM started working on a prototype system loosely based on Codd's concepts as *System R* in the early 1970s. The first version was ready in 1974/5, and work then started on multi-table systems in which the data could be split so that all of the data for a record (some of which is optional) did not have to be stored in a single large "chunk". Subsequent multi-user versions were tested by customers in 1978 and 1979, by which time a standardized query language € SQL[*citation needed*] € had been added. Codd's ideas were establishing themselves as both workable and superior to Codasyl, pushing IBM to develop a true production version of System R, known as *SQL/DS*, and, later, *Database 2* (DB2).

Larry Ellison's Oracle started from a different chain, based on IBM's papers on System R, and beat IBM to market when the first version was released in 1978.[*citation needed*]

Stonebraker went on to apply the lessons from INGRES to develop a new database, Postgres, which is now known as PostgreSQL. PostgreSQL is often used for global mission critical applications (the .org and .info domain name registries use it as their primary data store, as do many large companies and financial institutions).

In Sweden, Codd's paper was also read and Mimer SQL was developed from the mid-1970s at Uppsala University. In 1984, this project was consolidated into an independent enterprise. In the early 1980s, Mimer introduced transaction handling for high robustness in applications, an idea that was subsequently implemented on most other DBMS.

Another data model, the entity-relationship model, emerged in 1976 and gained popularity for database design as it emphasized a more familiar description than the earlier relational model. Later on, entity-relationship constructs were retrofitted as a data modeling construct for the relational model, and the difference between the two have become irrelevant.[*citation needed*]

1980s desktop databases

The 1980s ushered in the age of desktop computing. The new computers empowered their users with spreadsheets like Lotus 1,2,3 and database software like dBASE. The dBASE product was lightweight and easy for any computer user to understand out of the box. C. Wayne Ratliff the creator of dBASE stated: •dBASE was different from programs like BASIC, C, FORTRAN, and COBOL in that a lot of the dirty work had already been done. The data manipulation is done by dBASE instead of by the user, so the user can concentrate on what he is doing, rather than having to mess with the dirty details of opening, reading, and closing files, and managing space allocation.• [12] dBASE was one of the top selling software titles in the 1980s and early 1990s.

1980s object-oriented databases

The 1980s, along with a rise in object oriented programming, saw a growth in how data in various databases were handled. Programmers and designers began to treat the data in their databases as objects. That is to say that if a person's data were in a database, that person's attributes, such as their address, phone number, and age, were now considered to belong to that person instead of being extraneous data. This allows for relations between data to be relations to objects and their attributes and not to individual fields.[13] The term "object-relational impedance mismatch" described the inconvenience of translating between programmed objects and database tables. Object databases and object-relational databases attempt to solve this problem by providing an object-oriented language (sometimes as extensions to SQL) that programmers can use as alternative to purely relational SQL. On the programming side, libraries known as object-relational mappings (ORMs) attempt to solve the same problem.

2000s NoSQL and NewSQL databases

The next generation of post-relational databases in the 2000s became known as NoSQL databases, including fast key-value stores and document-oriented databases. XML databases are a type of structured document-oriented database that allows querying based on XML document attributes.

NoSQL databases are often very fast, do not require fixed table schemas, avoid join operations by storing denormalized data, and are designed to scale horizontally.

In recent years there was a high demand for massively distributed databases with high partition tolerance but according to the CAP theorem it is impossible for a distributed system to simultaneously provide consistency, availability and partition tolerance guarantees. A distributed system can satisfy any two of these guarantees at the same time, but not all three. For that reason many NoSQL databases are using what is called eventual consistency to provide both availability and partition tolerance guarantees with a maximum level of data consistency.

The most popular NoSQL systems include: MongoDB, Riak, Oracle NoSQL Database, memcached, Redis, CouchDB, Hazelcast, Apache Cassandra and HBase, note that all are open-source software products.

A number of new relational databases continuing use of SQL but aiming for performance comparable to NoSQL are known as NewSQL.

Database research

Database technology has been an active research topic since the 1960s, both in academia and in the research and development groups of companies (for example IBM Research). Research activity includes theory and development of prototypes. Notable research topics have included models, the atomic transaction concept and related concurrency control techniques, query languages and query optimization methods, RAID, and more.

The database research area has several dedicated academic journals (for example, ACM Transactions on Database Systems-TODS, Data and Knowledge Engineering-DKE) and annual conferences (e.g., ACM SIGMOD, ACM PODS, VLDB, IEEE ICDE).

Database type examples

One way to classify databases involves the type of their contents, for example: bibliographic, document-text, statistical, or multimedia objects. Another way is by their application area, for example: accounting, music compositions, movies, banking, manufacturing, or insurance. A third way is by some technical aspect, such as the database structure or interface type. This section lists a few of the adjectives used to characterize different kinds of databases.

- An in-memory database is a database that primarily resides in main memory, but is typically backed-up by non-volatile computer data storage. Main memory databases are faster than disk databases, and so are often used where response time is critical, such as in telecommunications network equipment.SAP HANA platform is a very hot topic for in-memory database. By May 2012, HANA was able to run on servers with 100TB main memory powered by IBM. The co founder of the company claimed that the system was big enough to run the 8 largest SAP customers.
- An active database includes an event-driven architecture which can respond to conditions both inside and outside the database. Possible uses include security monitoring, alerting, statistics gathering and authorization. Many databases provide active database features in the form of database triggers.
- A cloud database relies on cloud technology. Both the database and most of its DBMS reside remotely, "in the cloud," while its applications are both developed by programmers and later maintained and utilized by (application's) end-users through a web browser and Open APIs.
- Data warehouses archive data from operational databases and often from external sources such as market research firms. The warehouse becomes the central source of data for use by managers and other end-users who may not have access to operational data. For example, sales data might be aggregated to weekly totals and converted from internal product codes to use UPCs so that they can be compared with ACNielsen data. Some basic and essential components of data warehousing include retrieving, analyzing, and mining data, transforming, loading and managing data so as to make them available for further use.
- A deductive database combines logic programming with a relational database, for example by using the Datalog language.
- A distributed database is one in which both the data and the DBMS span multiple computers.
- A document-oriented database is designed for storing, retrieving, and managing document-oriented, or semi structured data, information. Document-oriented databases are one of the main categories of NoSQL databases.
- An embedded database system is a DBMS which is tightly integrated with an application software that requires access to stored data in such a way that the DBMS is hidden from the application,s end-users and requires little or no ongoing maintenance.[14]
- **End-user databases** consist of data developed by individual end-users. Examples of these are collections of documents, spreadsheets, presentations, multimedia, and other files. Several products exist to support such databases. Some of them are much simpler than full fledged DBMSs, with more elementary DBMS functionality.

- A federated database system comprises several distinct databases, each with its own DBMS. It is handled as a single database by a federated database management system (FDBMS), which transparently integrates multiple autonomous DBMSs, possibly of different types (in which case it would also be a heterogeneous database system), and provides them with an integrated conceptual view.
- Sometimes the term *multi-database* is used as a synonym to federated database, though it may refer to a less integrated (e.g., without an FDBMS and a managed integrated schema) group of databases that cooperate in a single application. In this case typically middleware is used for distribution, which typically includes an atomic commit protocol (ACP), e.g., the two-phase commit protocol, to allow distributed (global) transactions across the participating databases.
- A graph database is a kind of NoSQL database that uses graph structures with nodes, edges, and properties to represent and store information. General graph databases that can store any graph are distinct from specialized graph databases such as triplestores and network databases.
- In a hypertext or hypermedia database, any word or a piece of text representing an object, e.g., another piece of text, an article, a picture, or a film, can be hyperlinked to that object. Hypertext databases are particularly useful for organizing large amounts of disparate information. For example, they are useful for organizing online encyclopedias, where users can conveniently jump around the text. The World Wide Web is thus a large distributed hypertext database.
- A knowledge base (abbreviated **KB**, **kb** or •[15]) is a special kind of database for knowledge management, providing the means for the computerized collection, organization, and retrieval of knowledge. Also a collection of data representing problems with their solutions and related experiences.
- A mobile database can be carried on or synchronized from a mobile computing device.
- Operational databases store detailed data about the operations of an organization. They typically process relatively high volumes of updates using transactions. Examples include customer databases that record contact, credit, and demographic information about a business' customers, personnel databases that hold information such as salary, benefits, skills data about employees, enterprise resource planning systems that record details about product components, parts inventory, and financial databases that keep track of the organization's money, accounting and financial dealings.
- A parallel database seeks to improve performance through parallelization for tasks such as loading data, building indexes and evaluating queries.

  The major parallel DBMS architectures which are induced by the underlying hardware architecture are:

  - **Shared memory architecture**, where multiple processors share the main memory space, as well as other data storage.
  - **Shared disk architecture**, where each processing unit (typically consisting of multiple processors) has its own main memory, but all units share the other storage.
  - **Shared nothing architecture**, where each processing unit has its own main memory and other storage.
- Probabilistic databases employ fuzzy logic to draw inferences from imprecise data.
- Real-time databases process transactions fast enough for the result to come back and be acted on right away.
- A spatial database can store the data with multidimensional features. The queries on such data include location based queries, like "Where is the closest hotel in my area?".
- A temporal database has built-in time aspects, for example a temporal data model and a temporal version of SQL. More specifically the temporal aspects usually include valid-time and transaction-time.
- A terminology-oriented database builds upon an object-oriented database, often customized for a specific field.
- An unstructured data database is intended to store in a manageable and protected way diverse objects that do not fit naturally and conveniently in common databases. It may include email messages, documents, journals,

multimedia objects, etc. The name may be misleading since some objects can be highly structured. However, the entire possible object collection does not fit into a predefined structured framework. Most established DBMSs now support unstructured data in various ways, and new dedicated DBMSs are emerging.

Database design and modeling

The first task of a database designer is to produce a conceptual data model that reflects the structure of the information to be held in the database. A common approach to this is to develop an entity-relationship model, often with the aid of drawing tools. Another popular approach is the Unified Modeling Language. A successful data model will accurately reflect the possible state of the external world being modeled: for example, if people can have more than one phone number, it will allow this information to be captured. Designing a good conceptual data model requires a good understanding of the application domain; it typically involves asking deep questions about the things of interest to an organisation, like "can a customer also be a supplier?", or "if a product is sold with two different forms of packaging, are those the same product or different products?", or "if a plane flies from New York to Dubai via Frankfurt, is that one flight or two (or maybe even three)?". The answers to these questions establish definitions of the terminology used for entities (customers, products, flights, flight segments) and their relationships and attributes.

Producing the conceptual data model sometimes involves input from business processes, or the analysis of workflow in the organization. This can help to establish what information is needed in the database, and what can be left out. For example, it can help when deciding whether the database needs to hold historic data as well as current data.

Having produced a conceptual data model that users are happy with, the next stage is to translate this into a schema that implements the relevant data structures within the database. This process is often called logical database design, and the output is a logical data model expressed in the form of a schema. Whereas the conceptual data model is (in theory at least) independent of the choice of database technology, the logical data model will be expressed in terms of a particular database model supported by the chosen DBMS. (The terms *data model* and *database model* are often used interchangeably, but in this article we use *data model* for the design of a specific database, and *database model* for the modelling notation used to express that design.)

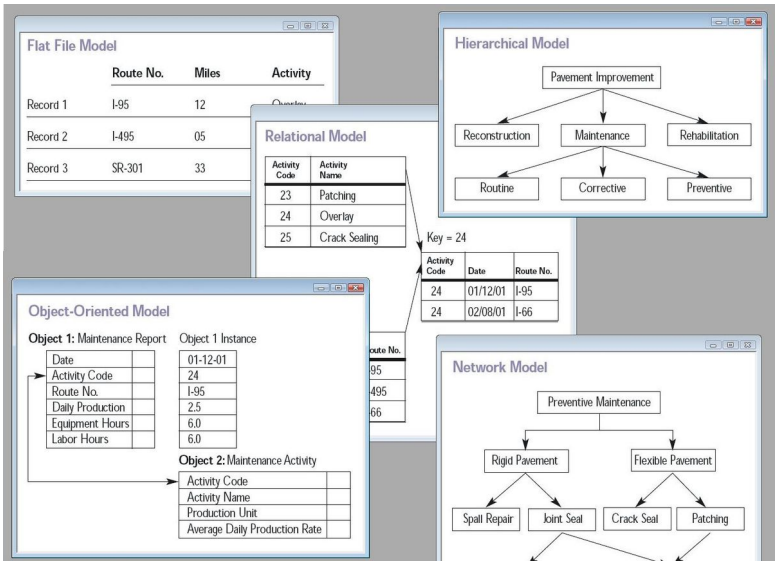
The most popular database model for general-purpose databases is the relational model, or more precisely, the relational model as represented by the SQL language. The process of creating a logical database design using this model uses a methodical approach known as normalization. The goal of normalization is to ensure that each elementary "fact" is only recorded in one place, so that insertions, updates, and deletions automatically maintain consistency.

The final stage of database design is to make the decisions that affect performance, scalability, recovery, security, and the like. This is often called *physical database design*. A key goal during this stage is data independence, meaning that the decisions made for performance optimization purposes should be invisible to end-users and applications. Physical design is driven mainly by performance requirements, and requires a good knowledge of the expected workload and access patterns, and a deep understanding of the features offered by the chosen DBMS.

Another aspect of physical database design is security. It involves both defining access control to database objects as well as defining security levels and methods for the data itself.

Database models

A database model is a type of data model that determines the logical structure of a database and fundamentally determines in which manner data can be stored, organized, and manipulated. The most popular example of a database model is the relational model (or the SQL approximation of relational), which uses a table-based format.



Collage of five types of database models.

Common logical data models for databases include:

- Hierarchical database model
- Network model
- Relational model
- Entity€relationship model
  - Enhanced entity€relationship model
- Object model
- Document model
- Entity€attribute€value model
- Star schema

An object-relational database combines the two related structures.

Physical data models include:

- Inverted index
- Flat file

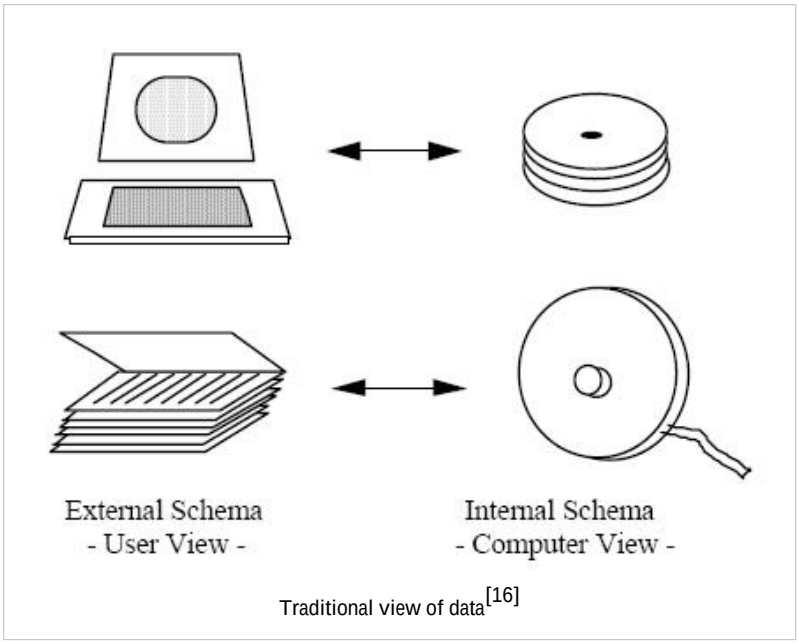
Other models include:

- Associative model
- Multidimensional model
- Multivalue model
- Semantic model
- XML database
- Named graph

External, conceptual, and internal views

A database management system provides three views of the database data:

- The **external level** defines how each group of end-users sees the organization of data in the database. A single database can have any number of views at the external level.
- The **conceptual level** unifies the various external views into a compatible global view. It provides the synthesis of all the external views. It is out of the scope of the various database end-users, and is rather of interest to database application developers and database administrators.
- The **internal level** (or *physical level*) is the internal organization of data inside a DBMS (see Implementation section below). It is concerned with cost, performance, scalability and other operational matters. It deals with storage layout of the data, using storage structures such as indexes to enhance performance. Occasionally it stores data of individual views (materialized views), computed from generic data, if performance justification exists for such redundancy. It balances all the external views' performance requirements, possibly conflicting, in an attempt to optimize overall performance across all activities.



Traditional view of data[16]

While there is typically only one conceptual (or logical) and physical (or internal) view of the data, there can be any number of different external views. This allows users to see database information in a more business-related way rather than from a technical, processing viewpoint. For example, a financial department of a company needs the payment details of all employees as part of the company's expenses, but does not need details about employees that are the interest of the human resources department. Thus different departments need different *views* of the company's database.

The three-level database architecture relates to the concept of *data independence* which was one of the major initial driving forces of the relational model. The idea is that changes made at a certain level do not affect the view at a higher level. For example, changes in the internal level do not affect application programs written using conceptual level interfaces, which reduces the impact of making physical changes to improve performance.

The conceptual view provides a level of indirection between internal and external. On one hand it provides a common view of the database, independent of different external view structures, and on the other hand it abstracts away details of how the data is stored or managed (internal level). In principle every level, and even every external view, can be presented by a different data model. In practice usually a given DBMS uses the same data model for both the external and the conceptual levels (e.g., relational model). The internal level, which is hidden inside the DBMS and depends on its implementation (see Implementation section below), requires a different level of detail and uses its own types of data structure types.

Separating the *external*, *conceptual* and *internal* levels was a major feature of the relational database model implementations that dominate 21st century databases.

Database languages

Database languages are special-purpose languages, which do one or more of the following:

- Data definition language - defines data types and the relationships among them
- Data manipulation language - performs tasks such as inserting, updating, or deleting data occurrences
- Query language - allows searching for information and computing derived information

Database languages are specific to a particular data model. Notable examples include:

- SQL combines the roles of data definition, data manipulation, and query in a single language. It was one of the first commercial languages for the relational model, although it departs in some respects from the relational model as described by Codd (for example, the rows and columns of a table can be ordered). SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standards (ISO) in 1987. The standards have been regularly enhanced since and is supported (with varying degrees of conformance) by all mainstream commercial relational DBMSs.
- OQL is an object model language standard (from the Object Data Management Group). It has influenced the design of some of the newer query languages like JDOQL and EJB QL.
- XQuery is a standard XML query language implemented by XML database systems such as MarkLogic and eXist, by relational databases with XML capability such as Oracle and DB2, and also by in-memory XML processors such as Saxon.
- SQL/XML combines XQuery with SQL.

A database language may also incorporate features like:

- DBMS-specific Configuration and storage engine management
- Computations to modify query results, like counting, summing, averaging, sorting, grouping, and cross-referencing
- Constraint enforcement (e.g. in an automotive database, only allowing one engine type per car)
- Application programming interface version of the query language, for programmer convenience

Performance, security, and availability

Because of the critical importance of database technology to the smooth running of an enterprise, database systems include complex mechanisms to deliver the required performance, security, and availability, and allow database administrators to control the use of these features.

Database storage

Database storage is the container of the physical materialization of a database. It comprises the *internal* (physical) *level* in the database architecture. It also contains all the information needed (e.g., metadata, "data about the data", and internal data structures) to reconstruct the *conceptual level* and *external level* from the internal level when needed. Putting data into permanent storage is generally the responsibility of the database engine a.k.a. "storage engine". Though typically accessed by a DBMS through the underlying operating system (and often utilizing the operating systems' file systems as intermediates for storage layout), storage properties and configuration setting are extremely important for the efficient operation of the DBMS, and thus are closely maintained by database administrators. A DBMS, while in operation, always has its database residing in several types of storage (e.g., memory and external storage). The database data and the additional needed information, possibly in very large amounts, are coded into bits. Data typically reside in the storage in structures that look completely different from the way the data look in the conceptual and external levels, but in ways that attempt to optimize (the best possible) these levels' reconstruction when needed by users and programs, as well as for computing additional types of needed information from the data (e.g., when querying the database).

Some DBMS support specifying which character encoding was used to store data, so multiple encodings can be used in the same database.

Various low-level database storage structures are used by the storage engine to serialize the data model so it can be written to the medium of choice. Techniques such as indexing may be used to improve performance. Conventional storage is row-oriented, but there are also column-oriented and correlation databases.

Database materialized views

Often storage redundancy is employed to increase performance. A common example is storing *materialized views*, which consist of frequently needed *external views* or query results. Storing such views saves the expensive computing of them each time they are needed. The downsides of materialized views are the overhead incurred when updating them to keep them synchronized with their original updated database data, and the cost of storage redundancy.

Database and database object replication

Occasionally a database employs storage redundancy by database objects replication (with one or more copies) to increase data availability (both to improve performance of simultaneous multiple end-user accesses to a same database object, and to provide resiliency in a case of partial failure of a distributed database). Updates of a replicated object need to be synchronized across the object copies. In many cases the entire database is replicated.

Database security

Database security deals with all various aspects of protecting the database content, its owners, and its users. It ranges from protection from intentional unauthorized database uses to unintentional database accesses by unauthorized entities (e.g., a person or a computer program).

Database access control deals with controlling who (a person or a certain computer program) is allowed to access what information in the database. The information may comprise specific database objects (e.g., record types, specific records, data structures), certain computations over certain objects (e.g., query types, or specific queries), or utilizing specific access paths to the former (e.g., using specific indexes or other data structures to access information). Database access controls are set by special authorized (by the database owner) personnel that uses dedicated protected security DBMS interfaces.

This may be managed directly on an individual basis, or by the assignment of individuals and privileges to groups, or (in the most elaborate models) through the assignment of individuals and groups to roles which are then granted entitlements. Data security prevents unauthorized users from viewing or updating the database. Using passwords, users are allowed access to the entire database or subsets of it called "subschemas". For example, an employee database can contain all the data about an individual employee, but one group of users may be authorized to view only payroll data, while others are allowed access to only work history and medical data. If the DBMS provides a way to interactively enter and update the database, as well as interrogate it, this capability allows for managing personal databases.

Data security in general deals with protecting specific chunks of data, both physically (i.e., from corruption, or destruction, or removal; e.g., see physical security), or the interpretation of them, or parts of them to meaningful information (e.g., by looking at the strings of bits that they comprise, concluding specific valid credit-card numbers; e.g., see data encryption).

Change and access logging records who accessed which attributes, what was changed, and when it was changed. Logging services allow for a forensic database audit later by keeping a record of access occurrences and changes. Sometimes application-level code is used to record changes rather than leaving this to the database. Monitoring can be set up to attempt to detect security breaches.

Transactions and concurrency

Database transactions can be used to introduce some level of fault tolerance and data integrity after recovery from a crash. A database transaction is a unit of work, typically encapsulating a number of operations over a database (e.g., reading a database object, writing, acquiring lock, etc.), an abstraction supported in database and also other systems. Each transaction has well defined boundaries in terms of which program/code executions are included in that transaction (determined by the transaction's programmer via special transaction commands).

The acronym ACID describes some ideal properties of a database transaction: Atomicity, Consistency, Isolation, and Durability.

Migration

See also section Database migration in article Data migration

A database built with one DBMS is not portable to another DBMS (i.e., the other DBMS cannot run it). However, in some situations it is desirable to move, migrate a database from one DBMS to another. The reasons are primarily economical (different DBMSs may have different total costs of ownership or TCOs), functional, and operational (different DBMSs may have different capabilities). The migration involves the database's transformation from one DBMS type to another. The transformation should maintain (if possible) the database related application (i.e., all related application programs) intact. Thus, the database's conceptual and external architectural levels should be maintained in the transformation. It may be desired that also some aspects of the architecture internal level are maintained. A complex or large database migration may be a complicated and costly (one-time) project by itself, which should be factored into the decision to migrate. This in spite of the fact that tools may exist to help migration between specific DBMS. Typically a DBMS vendor provides tools to help importing databases from other popular DBMSs.

Database building, maintaining, and tuning

After designing a database for an application arrives the stage of building the database. Typically an appropriate general-purpose DBMS can be selected to be utilized for this purpose. A DBMS provides the needed user interfaces to be utilized by database administrators to define the needed application's data structures within the DBMS's respective data model. Other user interfaces are used to select needed DBMS parameters (like security related, storage allocation parameters, etc.).

When the database is ready (all its data structures and other needed components are defined) it is typically populated with initial application's data (database initialization, which is typically a distinct project; in many cases using specialized DBMS interfaces that support bulk insertion) before making it operational. In some cases the database becomes operational while empty from application's data, and data are accumulated along its operation.

After completing building the database and making it operational arrives the database maintenance stage: Various database parameters may need changes and tuning for better performance, application's data structures may be changed or added, new related application programs may be written to add to the application's functionality, etc. Contribution by Malebye Joyce as adapted from informations systems for businesses from chapter 5 - storing ad organizing data. Databases are often confused with spread sheet such as Microsoft excel which is different from Microsoft access. Both can be used to store information,however a database serves a better function at this. Below is a comparison of spreadsheets and databases. Spread sheets strengths -1. Very simple data storage 2. Relatively easy to use 3. Require less planning Weaknesses- 1. Data integrity problems, include inaccurate,inconsistent and out of date version and out of date data. 2. Formulas could be incorrect Databases strengths 1. Methods for keeping data up to date and consistent 2. Data is of higher quality than data stored in spreadsheets 3. Good for storing and organizing information. Weakness 1. Require more planning and designing

Backup and restore

Sometimes it is desired to bring a database back to a previous state (for many reasons, e.g., cases when the database is found corrupted due to a software error, or if it has been updated with erroneous data). To achieve this a **backup** operation is done occasionally or continuously, where each desired database state (i.e., the values of its data and their embedding in database's data structures) is kept within dedicated backup files (many techniques exist to do this effectively). When this state is needed, i.e., when it is decided by a database administrator to bring the database back to this state (e.g., by specifying this state by a desired point in time when the database was in this state), these files are utilized to **restore** that state.

Other

Other DBMS features might include:

- Database logs
- Graphics component for producing graphs and charts, especially in a data warehouse system
- **Query optimizer** - Performs query optimization on every query to choose for it the most efficient *query plan* (a partial order (tree) of operations) to be executed to compute the query result. May be specific to a particular storage engine.
- Tools or hooks for database design, application programming, application program maintenance, database performance analysis and monitoring, database configuration monitoring, DBMS hardware configuration (a DBMS and related database may span computers, networks, and storage units) and related database mapping (especially for a distributed DBMS), storage allocation and database layout monitoring, storage migration, etc.

References

[1] Jeffrey Ullman 1997: *First course in database systems*, Prentice-Hall Inc., Simon & Schuster, Page 1, ISBN 0-13-861337-0.

[2] Tsitchizris, D. C. and F. H. Lochovsky (1982). *Data Models.* Englewood-Cliffs, Prentice-Hall.

[3] Beynon-Davies P. (2004). *Database Systems* 3rd Edition. Palgrave, Basingstoke, UK. ISBN 1-4039-1601-2

[4] . This article quotes a development time of 5 years involving 750 people for DB2 release 9 alone

[5] (Turing Award Lecture 1973)

[6] Codd, E.F. (1970). "A Relational Model of Data for Large Shared Data Banks" (http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf). In: *Communications of the ACM* 13 (6): 377€387.

[7] William Hershey and Carol Easthope, "A set theoretic data structure and retrieval language" (https://docs.google.com/open?id=0B4t_NX-QeWDYNmVhYjAwMWMtYzc3ZS00YjI0LWJhMjgtZTYyODZmNmFkNThh), Spring Joint Computer Conference, May 1972 in *ACM SIGIR Forum*, Volume 7, Issue 4 (December 1972), pp. 45-55, DOI= 10.1145/1095495.1095500 (http://doi.acm.org/10.1145/1095495.1095500)

[8] Ken North, "Sets, Data Models and Data Independence" (http://drdobbs.com/blogs/database/228700616), *Dr. Dobb's*, 10 March 2010

[9] *Description of a set-theoretic data structure* (http://hdl.handle.net/2027.42/4163), D. L. Childs, 1968, Technical Report 3 of the CONCOMP (Research in Conversational Use of Computers) Project, University of Michigan, Ann Arbor, Michigan, USA

[10] *Feasibility of a Set-Theoretic Data Structure : A General Structure Based on a Reconstituted Definition of Relation* (http://hdl.handle.net/2027.42/4164), D. L. Childs, 1968, Technical Report 6 of the CONCOMP (Research in Conversational Use of Computers) Project, University of Michigan, Ann Arbor, Michigan, USA

[11] *MICRO Information Management System (Version 5.0) Reference Manual* (http://docs.google.com/viewer?a=v&pid=explorer&chrome=true&srcid=0B4t_NX-QeWDYZGMwOTRmOTItZTg2Zi00YmJkLTg4MTktN2E4MWU0YmZlMjE3), M.A. Kahn, D.L. Rumelhart, and B.L. Bronson, October 1977, Institute of Labor and Industrial Relations (ILIR), University of Michigan and Wayne State University

[12] Interview with Wayne Ratliff (http://www.foxprohistory.org/interview_wayne_ratliff.htm). The FoxPro History. Retrieved on 2013-07-12.

[13] Development of an object-oriented DBMS; Portland, Oregon, United States; Pages: 472 € 482; 1986; ISBN 0-89791-204-7

[14] Graves, Steve. "COTS Databases For Embedded Systems" (http://www.embedded-computing.com/articles/id/?2020), *Embedded Computing Design* magazine, January 2007. Retrieved on August 13, 2008.

[15] Argumentation in Artificial Intelligence by Iyad Rahwan, Guillermo R. Simari

[16] itl.nist.gov (1993) *Integration Definition for Information Modeling (IDEFIX)* (http://www.itl.nist.gov/fipspubs/idef1x.doc). 21 December 1993.

Further reading

- Ling Liu and Tamer M. , zsu (Eds.) (2009). " Encyclopedia of Database Systems (http://www.springer.com/computer/database+management+&+information+retrieval/book/978-0-387-49616-0), 4100 p.ƒ60 illus. ISBN 978-0-387-49616-0.
- Beynon-Davies, P. (2004). Database Systems. 3rd Edition. Palgrave, Houndmills, Basingstoke.
- Connolly, Thomas and Carolyn Begg. *Database Systems.* New York: Harlow, 2002.
- Date, C. J. (2003). *An Introduction to Database Systems, Fifth Edition*. Addison Wesley. ISBNƒ0-201-51381-1.
- Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*, 1st edition, Morgan Kaufmann Publishers, 1992.
- Kroenke, David M. and David J. Auer. *Database Concepts.* 3rd ed. New York: Prentice, 2007.
- Raghu Ramakrishnan and Johannes Gehrke, *Database Management Systems* (http://pages.cs.wisc.edu/~dbbook/)
- Abraham Silberschatz, Henry F. Korth, S. Sudarshan, *Database System Concepts* (http://www.db-book.com/)
- Discussion on database systems, (http://www.bbconsult.co.uk/Documents/Database-Systems.docx)
- Lightstone, S.; Teorey, T.; Nadeau, T. (2007). *Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more*. Morgan Kaufmann Press. ISBNƒ0-12-369389-6.
- Teorey, T.; Lightstone, S. and Nadeau, T. *Database Modeling & Design: Logical Design*, 4th edition, Morgan Kaufmann Press, 2005. ISBN 0-12-685352-5

External links

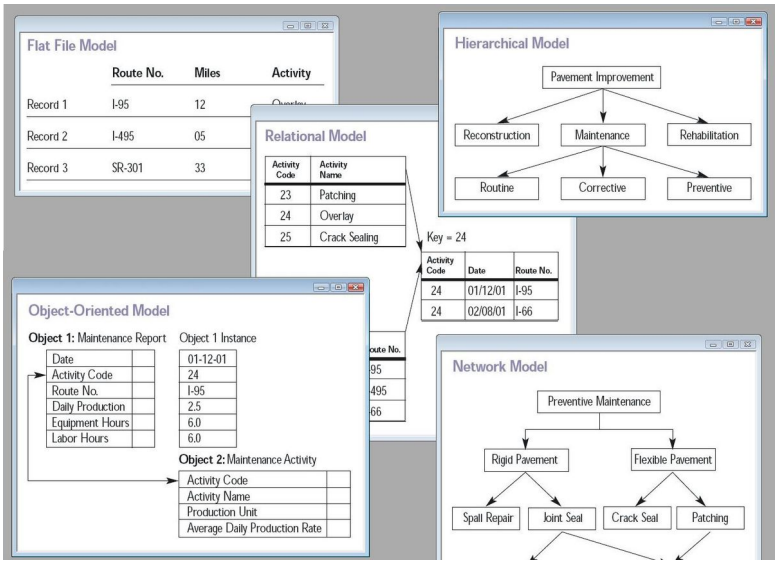
- Database (http://www.dmoz.org/Computers/Data_Formats/Database/) at the Open Directory Project

Database model

A **database model** is a type of data model that determines the logical structure of a database and fundamentally determines in which manner data can be stored, organized, and manipulated. The most popular example of a database model is the relational model, which uses a table-based format.

Common logical data models for databases include:

- Hierarchical database model
- Network model
- Relational model
- Entity€relationship model
  - Enhanced entity€relationship model
- Object model
- Document model
- Entity€attribute€value model



Collage of five types of database models.

- Star schema

An object-relational database combines the two related structures.

Physical data models include:

- Inverted index
- Flat file

Other models include:

- Associative model
- Multidimensional model
- Multivalue model
- Semantic model
- XML database
- Named graph
- Triplestore

Relationships and functions

A given database management system may provide one or more of the five models. The optimal structure depends on the natural organization of the application's data, and on the application's requirements, which include transaction rate (speed), reliability, maintainability, scalability, and cost. Most database management systems are built around one particular data model, although it is possible for products to offer support for more than one model.

Various physical data models can implement any given logical model. Most database software will offer the user some level of control in tuning the physical implementation, since the choices that are made have a significant effect on performance.

A model is not just a way of structuring data: it also defines a set of operations that can be performed on the data. The relational model, for example, defines operations such as select (project) and join. Although these operations may not be explicit in a particular query language, they provide the foundation on which a query language is built.

Flat model

The flat (or table) model consists of a single, two-dimensional array of data elements, where all members of a given column are assumed to be similar values, and all members of a row are assumed to be related to one another. For instance, columns for name and password that might be used as a part of a system security database. Each row would have the specific password associated with an individual user. Columns of the table often have a type associated with them, defining them as character data, date or time information, integers, or floating point numbers. This tabular format is a precursor to the relational model.

Flat File Model

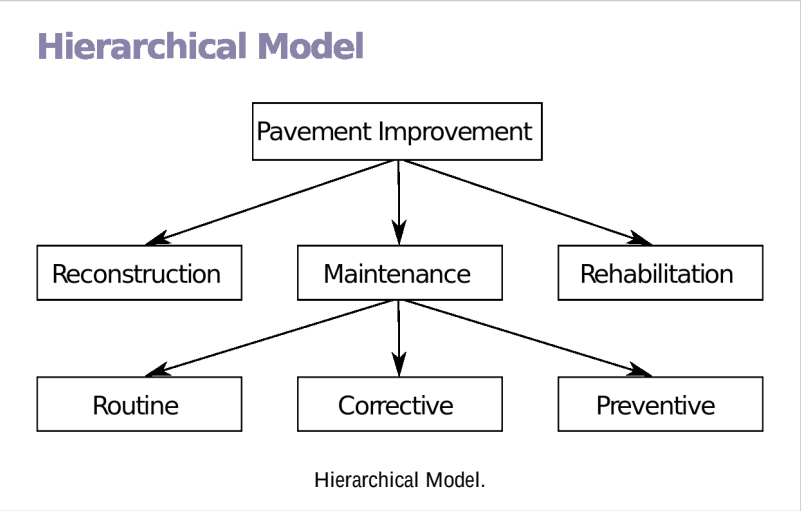
	Route No.	Miles	Activity
Record 1	I-95	12	Overlay
Record 2	I-495	05	Patching
Record 3	SR-301	33	Crack seal

Flat File Model.

Early data models

These models were popular in the 1960s, 1970s, but nowadays can be found primarily in old legacy systems. They are characterized primarily by being navigational with strong connections between their logical and physical representations, and deficiencies in data independence.

Hierarchical model

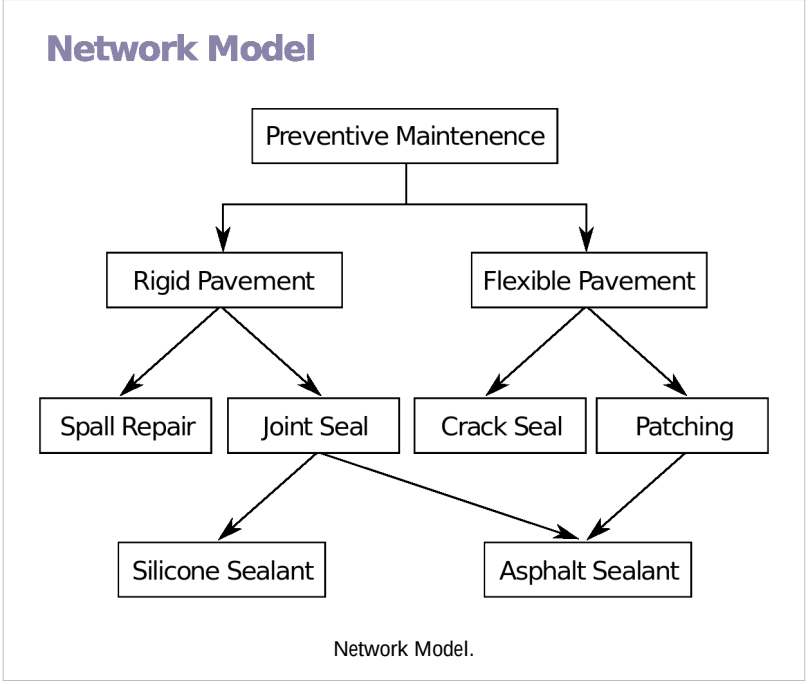


Hierarchical Model.

In a hierarchical model, data is organized into a tree-like structure, implying a single parent for each record. A sort field keeps sibling records in a particular order. Hierarchical structures were widely used in the early mainframe database management systems, such as the Information Management System (IMS) by IBM, and now describe the structure of XML documents. This structure allows one one-to-many relationship between two types of data. This structure is very efficient to describe many relationships in the real world; recipes, table of contents, ordering of paragraphs/verses, any nested and sorted information.

This hierarchy is used as the physical order of records in storage. Record access is done by navigating through the data structure using pointers combined with sequential accessing. Because of this, the hierarchical structure is inefficient for certain database operations when a full path (as opposed to upward link and sort field) is not also included for each record. Such limitations have been compensated for in later IMS versions by additional logical hierarchies imposed on the base physical hierarchy.

Network model



Network Model.

The network model expands upon the hierarchical structure, allowing many-to-many relationships in a tree-like structure that allows multiple parents. It was the most popular before being replaced by the relational model, and is defined by the CODASYL specification.

The network model organizes data using two fundamental concepts, called *records* and *sets*. Records contain fields (which may be organized hierarchically, as in the programming language COBOL). Sets (not to be confused with mathematical sets) define one-to-many[1] relationships between records: one owner, many members. A record may be an owner in any number of sets, and a member in any number of sets.

A set consists of circular linked lists where one record type, the set owner or parent, appears once in each circle, and a second record type, the subordinate or child, may appear multiple times in each circle. In this way a hierarchy may be established between any two record types, e.g., type A is the owner of B. At the same time another set may be defined where B is the owner of A. Thus all the sets comprise a general directed graph (ownership defines a direction), or *network* construct. Access to records is either sequential (usually in each record type) or by navigation in the circular linked lists.

The network model is able to represent redundancy in data more efficiently than in the hierarchical model, and there can be more than one path from an ancestor node to a descendant. The operations of the network model are navigational in style: a program maintains a current position, and navigates from one record to another by following the relationships in which the record participates. Records can also be located by supplying key values.

Although it is not an essential feature of the model, network databases generally implement the set relationships by means of pointers that directly address the location of a record on disk. This gives excellent retrieval performance, at the expense of operations such as database loading and reorganization.

Popular DBMS products that utilized it were Cincom Systems' Total and Cullinet's IDMS. IDMS gained a considerable customer base; in the 1980s, it adopted the relational model and SQL in addition to its original tools and languages.

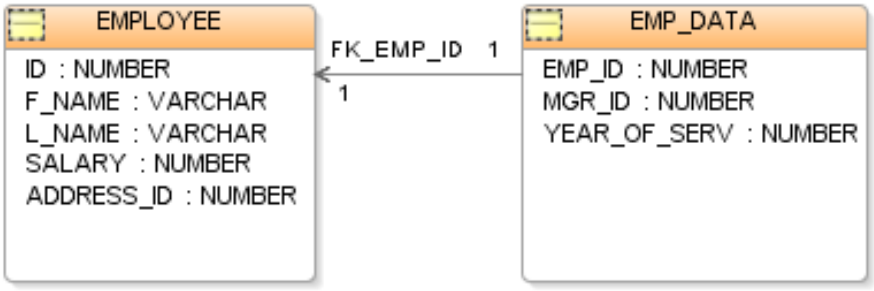
Most object databases (invented in the 1990s) use the navigational concept to provide fast navigation across networks of objects, generally using object identifiers as "smart" pointers to related objects. Objectivity/DB, for instance, implements named one-to-one, one-to-many, many-to-one, and many-to-many named relationships that can cross databases. Many object databases also support SQL, combining the strengths of both models.

Inverted file model

In an *inverted file* or *inverted index*, the contents of the data are used as keys in a lookup table, and the values in the table are pointers to the location of each instance of a given content item. This is also the logical structure of contemporary database indexes, which might only use the contents from a particular columns in the lookup table. The *inverted file data model* can put indexes in a second set of files next to existing flat database files, in order to efficiently directly access needed records in these files.

Notable for using this data model is the ADABAS DBMS of Software AG, introduced in 1970. ADABAS has gained considerable customer base and exists and supported until today. In the 1980s it has adopted the relational model and SQL in addition to its original tools and languages.

Relational model



The relational model was introduced by E.F. Codd in 1970[2] as a way to make database management systems more independent of any particular application. It is a mathematical model defined in terms of predicate logic and set theory, and systems implementing it have been used by mainframe, midrange and microcomputer systems.

The products that are generally referred to as relational databases in fact implement a model that is only an approximation to the mathematical model defined by Codd. Three key terms are used extensively in relational database models: *relations*, *attributes*, and *domains*. A relation is a table with columns and rows. The named columns of the relation are called attributes, and the domain is the set of values the attributes are allowed to take.

The basic data structure of the relational model is the table, where information about a particular entity (say, an employee) is represented in rows (also called tuples) and columns. Thus, the "relation" in "relational database" refers to the various tables in the database; a relation is a set of tuples. The columns enumerate the various attributes of the entity (the employee's name, address or phone number, for example), and a row is an actual instance of the entity (a specific employee) that is represented by the relation. As a result, each tuple of the employee table represents various attributes of a single employee.

All relations (and, thus, tables) in a relational database have to adhere to some basic rules to qualify as relations. First, the ordering of columns is immaterial in a table. Second, there can't be identical tuples or rows in a table. And third, each tuple will contain a single value for each of its attributes.

A relational database contains multiple tables, each similar to the one in the "flat" database model. One of the strengths of the relational model is that, in principle, any value occurring in two different records (belonging to the same table or to different tables), implies a relationship among those two records. Yet, in order to enforce explicit integrity constraints, relationships between records in tables can also be defined explicitly, by identifying or non-identifying parent-child relationships characterized by assigning cardinality (1:1, (0)1:M, M:M). Tables can also have a designated single attribute or a set of attributes that can act as a "key", which can be used to uniquely identify each tuple in the table.

A key that can be used to uniquely identify a row in a table is called a primary key. Keys are commonly used to join or combine data from two or more tables. For example, an *Employee* table may contain a column named *Location* which contains a value that matches the key of a *Location* table. Keys are also critical in the creation of indexes, which facilitate fast retrieval of data from large tables. Any column can be a key, or multiple columns can be

grouped together into a compound key. It is not necessary to define all the keys in advance; a column can be used as a key even if it was not originally intended to be one.

A key that has an external, real-world meaning (such as a person's name, a book's ISBN, or a car's serial number) is sometimes called a "natural" key. If no natural key is suitable (think of the many people named *Brown*), an arbitrary or surrogate key can be assigned (such as by giving employees ID numbers). In practice, most databases have both generated and natural keys, because generated keys can be used internally to create links between rows that cannot break, while natural keys can be used, less reliably, for searches and for integration with other databases. (For example, records in two independently developed databases could be matched up by social security number, except when the social security numbers are incorrect, missing, or have changed.)

The most common query language used with the relational model is the Structured Query Language (SQL).

Dimensional model

The dimensional model is a specialized adaptation of the relational model used to represent data in data warehouses in a way that data can be easily summarized using online analytical processing, or OLAP queries. In the dimensional model, a database schema consists of a single large table of facts that are described using dimensions and measures. A dimension provides the context of a fact (such as who participated, when and where it happened, and its type) and is used in queries to group related facts together. Dimensions tend to be discrete and are often hierarchical; for example, the location might include the building, state, and country. A measure is a quantity describing the fact, such as revenue. It is important that measures can be meaningfully aggregated *ƒ* for example, the revenue from different locations can be added together.

In an OLAP query, dimensions are chosen and the facts are grouped and aggregated together to create a summary.

The dimensional model is often implemented on top of the relational model using a star schema, consisting of one highly normalized table containing the facts, and surrounding denormalized tables containing each dimension. An alternative physical implementation, called a snowflake schema, normalizes multi-level hierarchies within a dimension into multiple tables.

A data warehouse can contain multiple dimensional schemas that share dimension tables, allowing them to be used together. Coming up with a standard set of dimensions is an important part of dimensional modeling.

Its high performance has made the dimensional model the most popular database structure for OLAP.

Post-relational database models

Products offering a more general data model than the relational model are sometimes classified as *post-relational*.[3] Alternate terms include "hybrid database", "Object-enhanced RDBMS" and others. The data model in such products incorporates relations but is not constrained by E.F. Codd's Information Principle, which requires that

> all information in the database must be cast explicitly in terms of values in relations and in no other way

Some of these extensions to the relational model integrate concepts from technologies that pre-date the relational model. For example, they allow representation of a directed graph with trees on the nodes. The German company *sones* implements this concept in its GraphDB.

Some post-relational products extend relational systems with non-relational features. Others arrived in much the same place by adding relational features to pre-relational systems. Paradoxically, this allows products that are historically pre-relational, such as PICK and MUMPS, to make a plausible claim to be post-relational.

The resource space model (RSM) is a non-relational data model based on multi-dimensional classification.

Graph model

Graph databases allow even more general structure than a network database; any node may be connected to any other node.

Multivalue model

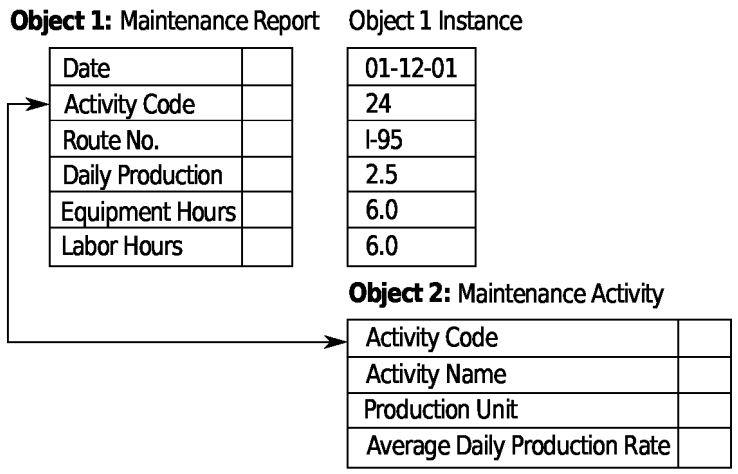
Multivalue databases are "lumpy" data, in that they can store exactly the same way as relational databases, but they also permit a level of depth which the relational model can only approximate using sub-tables. This is nearly identical to the way XML expresses data, where a given field/attribute can have multiple right answers at the same time. Multivalue can be thought of as a compressed form of XML.

An example is an invoice, which in either multivalue or relational data could be seen as (A) Invoice Header Table - one entry per invoice, and (B) Invoice Detail Table - one entry per line item. In the multivalue model, we have the option of storing the data as on table, with an embedded table to represent the detail: (A) Invoice Table - one entry per invoice, no other tables needed.

The advantage is that the atomicity of the Invoice (conceptual) and the Invoice (data representation) are one-to-one. This also results in fewer reads, less referential integrity issues, and a dramatic decrease in the hardware needed to support a given transaction volume.

Object-oriented database models

In the 1990s, the object-oriented programming paradigm was applied to database technology, creating a new database model known as object databases. This aims to avoid the object-relational impedance mismatch - the overhead of converting information between its representation in the database (for example as rows in tables) and its representation in the application program (typically as objects). Even further, the type system used in a particular application can be defined directly in the database, allowing the database to enforce the same data integrity invariants. Object databases also introduce the key ideas of object programming, such as encapsulation and polymorphism, into the world of databases.



Example of an Object-Oriented Model.

A variety of these ways have been tried Wikipedia:Manual of Style/Words to watch#Unsupported attributionsfor storing objects in a database. SomeWikipedia:Avoid weasel words products have approached the problem from the application programming end, by making the objects manipulated by the program persistent. This typically requires the addition of some kind of query language, since conventional programming languages do not have the ability to find objects based on their information content. OthersWikipedia:Avoid weasel words have attacked the problem from the database end, by defining an object-oriented data model for the database, and defining a database programming language that allows full programming capabilities as well as traditional query facilities.

Object databases suffered because of a lack of standardization: although standards were defined by ODMG, they were never implemented well enough to ensure interoperability between products. Nevertheless, object databases

have been used successfully in many applications: usually specialized applications such as engineering databases or molecular biology databases rather than mainstream commercial data processing. However, object database ideas were picked up by the relational vendors and influenced extensions made to these products and indeed to the SQL language.

An alternative to translating between objects and relational databases is to use an object-relational mapping (ORM) library.

References

[1] http://toolserver.org/%7Edispenser/cgi-bin/dab_solver.py?page=Database_model&editintro=Template:Disambiguation_needed/editintro&client=Template:Dn

[2] E.F. Codd (1970). "A relational model of data for large shared data banks". In: *Communications of the ACM archive*. Vol 13. Issue 6(June 1970). pp.377-387.

[3] *Introducing databases* by Stephen Chu, in Conrick, M. (2006) *Health informatics: transforming healthcare with technology*, Thomson, ISBN 0-17-012731-1, p. 69.

Database normalization

Database normalization is the process of organizing the fields and tables of a relational database to minimize redundancy and dependency. Normalization usually involves dividing large tables into smaller (and less redundant) tables and defining relationships between them. The objective is to isolate data so that additions, deletions, and modifications of a field can be made in just one table and then propagated through the rest of the database using the defined relationships.

Edgar F. Codd, the inventor of the relational model, introduced the concept of normalization and what we now know as the First Normal Form (1NF) in 1970. Codd went on to define the Second Normal Form (2NF) and Third Normal Form (3NF) in 1971,[1] and Codd and Raymond F. Boyce defined the Boyce-Codd Normal Form (BCNF) in 1974.[2] Informally, a relational database table is often described as "normalized" if it is in the Third Normal Form.[3] Most 3NF tables are free of insertion, update, and deletion anomalies.

A standard piece of database design guidance is that the designer should first create a fully normalized design; then selective denormalization can be performed for performance reasons.[4]

Objectives of normalization

A basic objective of the first normal form defined by Edgar Frank "Ted" Codd in 1970 was to permit data to be queried and manipulated using a "universal data sub-language" grounded in first-order logic.[5] (SQL is an example of such a data sub-language, albeit one that Codd regarded as seriously flawed.)[6]

The objectives of normalization beyond 1NF (First Normal Form) were stated as follows by Codd:

> 1. To free the collection of relations from undesirable insertion, update and deletion dependencies;
>
> 2. To reduce the need for restructuring the collection of relations, as new types of data are introduced, and thus increase the life span of application programs;
>
> 3. To make the relational model more informative to users;
>
> 4. To make the collection of relations neutral to the query statistics, where these statistics are liable to change as time goes by.
>
> ƒ E.F. Codd, *"Further Normalization of the Data Base Relational Model"*[7]

The sections below give details of each of these objectives.

Free the database of modification anomalies

When an attempt is made to modify (update, insert into, or delete from) a table, undesired side-effects may follow. Not all tables can suffer from these side-effects; rather, the side-effects can only arise in tables that have not been sufficiently normalized. An insufficiently normalized table might have one or more of the following characteristics:

Employees' Skills

Employee ID	Employee Address	Skill
426	87 Sycamore Grove	Typing
426	87 Sycamore Grove	Shorthand
519	94 Chestnut Street	Public Speaking
519	96 Walnut Avenue	Carpentry

An **update anomaly**. Employee 519 is shown as having different addresses on different records.

Faculty and Their Courses

Faculty ID	Faculty Name	Faculty Hire Date	Course Code
389	Dr. Giddens	10-Feb-1985	ENG-206
407	Dr. Saperstein	19-Apr-1999	CMP-101
407	Dr. Saperstein	19-Apr-1999	CMP-201
424	Dr. Newsome	29-Mar-2007	?

An **insertion anomaly**. Until the new faculty member, Dr. Newsome, is assigned to teach at least one course, his details cannot be recorded.

Faculty and Their Courses

Faculty ID	Faculty Name	Faculty Hire Date	Course Code
389	Dr. Giddens	10-Feb-1985	ENG-206
407	Dr. Saperstein	19-Apr-1999	CMP-101
407	Dr. Saperstein	19-Apr-1999	CMP-201

DELETE

A **deletion anomaly**. All information about Dr. Giddens is lost if he temporarily ceases to be assigned to any courses.

- The same information can be expressed on multiple rows; therefore updates to the table may result in logical inconsistencies. For example, each record in an "Employees' Skills" table might contain an Employee ID, Employee Address, and Skill; thus a change of address for a particular employee will potentially need to be applied to multiple records (one for each skill). If the update is not carried through successfully ƒ if, that is, the employee's address is updated on some records but not others ƒ then the table is left in an inconsistent state. Specifically, the table provides conflicting answers to the question of what this particular employee's address is. This phenomenon is known as an **update anomaly**.
- There are circumstances in which certain facts cannot be recorded at all. For example, each record in a "Faculty and Their Courses" table might contain a Faculty ID, Faculty Name, Faculty Hire Date, and Course Code ƒ thus we can record the details of any faculty member who teaches at least one course, but we cannot record the details of a newly hired faculty member who has not yet been assigned to teach any courses except by setting the Course Code to null. This phenomenon is known as an **insertion anomaly**.
- Under certain circumstances, deletion of data representing certain facts necessitates deletion of data representing completely different facts. The "Faculty and Their Courses" table described in the previous example suffers from this type of anomaly, for if a faculty member temporarily ceases to be assigned to any courses, we must delete the last of the records on which that faculty member appears, effectively also deleting the faculty member. This phenomenon is known as a **deletion anomaly**.

Minimize redesign when extending the database structure

When a fully normalized database structure is extended to allow it to accommodate new types of data, the pre-existing aspects of the database structure can remain largely or entirely unchanged. As a result, applications interacting with the database are minimally affected.

Make the data model more informative to users

Normalized tables, and the relationship between one normalized table and another, mirror real-world concepts and their interrelationships.

Avoid bias towards any particular pattern of querying

Normalized tables are suitable for general-purpose querying. This means any queries against these tables, including future queries whose details cannot be anticipated, are supported. In contrast, tables that are not normalized lend themselves to some types of queries, but not others.

For example, consider an online bookseller whose customers maintain wishlists of books they'd like to have. For the obvious, anticipated query *ƒ* what books does this customer want? *ƒ* it's enough to store the customer's wishlist in the table as, say, a homogeneous string of authors and titles.

With this design, though, the database can answer only that one single query. It cannot by itself answer interesting but unanticipated queries: What is the most-wished-for book? Which customers are interested in WWII espionage? How does Lord Byron stack up against his contemporary poets? Answers to these questions must come from special adaptive tools completely separate from the database. One tool might be software written especially to handle such queries. This special adaptive software has just one single purpose: in effect to normalize the non-normalized field.

Unforeseen queries can be answered trivially, and entirely within the database framework, with a normalized table.

Example

Querying and manipulating the data within a data structure which is not normalized, such as the following non-1NF representation of customers' credit card transactions, involves more complexity than is really necessary:

Customer Jones Wilkinson Stevens Transactions

Tr. ID	Date	Amount
12890	14-Oct-2003	„87
12904	15-Oct-2003	„50

Tr. ID	Date	Amount
12898	14-Oct-2003	„21

Tr. ID	Date	Amount
12907	15-Oct-2003	„18
14920	20-Nov-2003	„70
15003	27-Nov-2003	„60

To each customer corresponds a *repeating group* of transactions. The automated evaluation of any query relating to customers' transactions therefore would broadly involve two stages:

1. Unpacking one or more customers' groups of transactions allowing the individual transactions in a group to be examined, and
2. Deriving a query result based on the results of the first stage

For example, in order to find out the monetary sum of all transactions that occurred in October 2003 for all customers, the system would have to know that it must first unpack the *Transactions* group of each customer, then sum the *Amounts* of all transactions thus obtained where the *Date* of the transaction falls in October 2003.

One of Codd's important insights was that this structural complexity could always be removed completely, leading to much greater power and flexibility in the way queries could be formulated (by users and applications) and evaluated (by the DBMS). The normalized equivalent of the structure above would look like this:

Customer	Tr. ID	Date	Amount
Jones	12890	14-Oct-2003	„87
Jones	12904	15-Oct-2003	„50
Wilkins	12898	14-Oct-2003	„21
Stevens	12907	15-Oct-2003	„18
Stevens	14920	20-Nov-2003	„70
Stevens	15003	27-Nov-2003	„60

Now each row represents an individual credit card transaction, and the DBMS can obtain the answer of interest, simply by finding all rows with a Date falling in October, and summing their Amounts. The data structure places all of the values on an equal footing, exposing each to the DBMS directly, so each can potentially participate directly in queries; whereas in the previous situation some values were embedded in lower-level structures that had to be handled specially. Accordingly, the normalized design lends itself to general-purpose query processing, whereas the unnormalized design does not.

Background to normalization: definitions

Functional dependency

> In a given table, an attribute *Y* is said to have a functional dependency on a set of attributes *X* (written *X* … *Y*) if and only if each *X* value is associated with precisely one *Y* value. For example, in an "Employee" table that includes the attributes "Employee ID" and "Employee Date of Birth", the functional dependency {Employee ID} … {Employee Date of Birth} would hold. It follows from the previous two sentences that each {Employee ID} is associated with precisely one {Employee Date of Birth}.

Trivial functional dependency

> A trivial functional dependency is a functional dependency of an attribute on a superset of itself. {Employee ID, Employee Address} … {Employee Address} is trivial, as is {Employee Address} … {Employee Address}.

Full functional dependency

An attribute is fully functionally dependent on a set of attributes X if it is:

- functionally dependent on X, and
- not functionally dependent on any proper subset of X. {Employee Address} has a functional dependency on {Employee ID, Skill}, but not a *full* functional dependency, because it is also dependent on {Employee ID}.Even by the removal of {Skill} functional dependency still holds between {Employee Address} and {Employee ID}.

Transitive dependency

A transitive dependency is an indirect functional dependency, one in which *X*…*Z* only by virtue of *X*…*Y* and *Y*…*Z*.

Multivalued dependency

A multivalued dependency is a constraint according to which the presence of certain rows in a table implies the presence of certain other rows.

Join dependency

A table *T* is subject to a join dependency if *T* can always be recreated by joining multiple tables each having a subset of the attributes of *T*.

Superkey

A superkey is a combination of attributes that can be used to uniquely identify a database record. A table might have many superkeys.

Candidate key

A candidate key is a special subset of superkeys that do not have any extraneous information in them: it is a minimal superkey.

Example:

A table with the fields <Name>, <Age>, <SSN> and <Phone Extension> has many possible superkeys. Three of these are <SSN>, <Phone Extension, Name> and <SSN, Name>. Of those, only <SSN> is a candidate key as the others contain information not necessary to uniquely identify records ('SSN' here refers to Social Security Number, which is unique to each person).

Non-prime attribute

A non-prime attribute is an attribute that does not occur in any candidate key. Employee Address would be a non-prime attribute in the "Employees' Skills" table.

Prime attribute

A prime attribute, conversely, is an attribute that does occur in some candidate key.

Primary key

One candidate key in a relation may be designated the primary key. While that may be a common practice (or even a required one in some environments), it is strictly notational and has no bearing on normalization. With respect to normalization, all candidate keys have equal standing and are treated the same.

Normal forms

The **normal forms** (abbrev. **NF**) of relational database theory provide criteria for determining a table's degree of immunity against logical inconsistencies and anomalies. The higher the normal form applicable to a table, the less vulnerable it is. Each table has a "**highest normal form**" (**HNF**): by definition, a table always meets the requirements of its HNF and of all normal forms lower than its HNF; also by definition, a table fails to meet the requirements of any normal form higher than its HNF.

The normal forms are applicable to individual tables; to say that an entire database is in normal form *n* is to say that all of its tables are in normal form *n*.

Newcomers to database design sometimes suppose that normalization proceeds in an iterative fashion, i.e. a 1NF design is first normalized to 2NF, then to 3NF, and so on. This is not an accurate description of how normalization typically works. A sensibly designed table is likely to be in 3NF on the first attempt; furthermore, if it is 3NF, it is overwhelmingly likely to have an HNF of 5NF. Achieving the "higher" normal forms (above 3NF) does not usually require an extra expenditure of effort on the part of the designer, because 3NF tables usually need no modification to meet the requirements of these higher normal forms.

The main normal forms are summarized below.

	Normal form	Defined by	In	Brief definition
1NF	First normal form	Two versions: E.F. Codd (1970), C.J. Date (2003)	1970 and 2003 [8]	A relation is in first normal form if the domain of each attribute contains only atomic values, and the value of each attribute contains only a single value from that domain.
2NF	Second normal form	E.F. Codd	1971	No non-prime attribute in the table is functionally dependent on a proper subset of any candidate key
3NF	Third normal form	Two versions: E.F. Codd (1971), C. Zaniolo (1982)	1971 and 1982 [9]	Every non-prime attribute is non-transitively dependent on every candidate key in the table. The attributes that do not contribute to the description of the primary key are removed from the table. In other words, no transitive dependency is allowed.
EKNF	Elementary Key Normal Form	C. Zaniolo	1982	Every non-trivial functional dependency in the table is either the dependency of an elementary key attribute or a dependency on a superkey
BCNF	Boyce€Codd normal form	Raymond F. Boyce and E.F. Codd	1974 [10]	Every non-trivial functional dependency in the table is a dependency on a superkey
4NF	Fourth normal form	Ronald Fagin	1977	Every non-trivial multivalued dependency in the table is a dependency on a superkey
5NF	Fifth normal form	Ronald Fagin	1979 [11]	Every non-trivial join dependency in the table is implied by the superkeys of the table
DKNF	Domain/key normal form	Ronald Fagin	1981 [12]	Every constraint on the table is a logical consequence of the table's domain constraints and key constraints
6NF	Sixth normal form	C.J. Date, Hugh Darwen, and Nikos Lorentzos	2002 [13]	Table features no non-trivial join dependencies at all (with reference to generalized join operator)

Denormalization

Databases intended for online transaction processing (OLTP) are typically more normalized than databases intended for online analytical processing (OLAP). OLTP applications are characterized by a high volume of small transactions such as updating a sales record at a supermarket checkout counter. The expectation is that each transaction will leave the database in a consistent state. By contrast, databases intended for OLAP operations are primarily "read mostly" databases. OLAP applications tend to extract historical data that has accumulated over a long period of time. For such databases, redundant or "denormalized" data may facilitate business intelligence applications. Specifically, dimensional tables in a star schema often contain denormalized data. The denormalized or redundant data must be carefully controlled during extract, transform, load (ETL) processing, and users should not be permitted to see the data until it is in a consistent state. The normalized alternative to the star schema is the snowflake schema. In many cases, the need for denormalization has waned as computers and RDBMS software have become more powerful, but since data volumes have generally increased along with hardware and software

performance, OLAP databases often still use denormalized schemas.

Denormalization is also used to improve performance on smaller computers as in computerized cash-registers and mobile devices, since these may use the data for look-up only (e.g. price lookups). Denormalization may also be used when no RDBMS exists for a platform (such as Palm), or no changes are to be made to the data and a swift response is crucial.

Non-first normal form (NF€ or N1NF)

Denormalization is the opposite of normalization. In recognition that denormalization can be deliberate and useful, the non-first normal form is a definition of database designs which do not conform to first normal form, by allowing "sets and sets of sets to be attribute domains" (Schek 1982). The languages used to query and manipulate data in the model must be extended accordingly to support such values.

One way of looking at this is to consider such structured values as being specialized types of values (domains), with their own domain-specific languages. However, what is usually meant by non-1NF models is the approach in which the relational model and the languages used to query it are extended with a general mechanism for such structure; for instance, the nested relational model supports the use of relations as domain values, by adding two additional operators (*nest* and *unnest*) to the relational algebra that can create and flatten nested relations, respectively.

Consider the following table:

First Normal Form

Person	Favourite Colour
Bob	blue
Bob	red
Jane	green
Jane	yellow
Jane	red

Assume a person has several favourite colours. Obviously, favourite colours consist of a set of colours modeled by the given table. To transform a 1NF into an NF„ table a "nest" operator is required which extends the relational algebra of the higher normal forms. Applying the "nest" operator to the 1NF table yields the following NF„ table:

Non-First Normal Form

Person	Favourite Colours
Bob	**Favourite Colour** blue red
Jane	**Favourite Colour** green yellow red

To transform this NF„ table back into a 1NF an "unnest" operator is required which extends the relational algebra of the higher normal forms. The unnest, in this case, would make "colours" into its own table.

Although "unnest" is the mathematical inverse to "nest", the operator "nest" is not always the mathematical inverse of "unnest". Another constraint required is for the operators to be bijective, which is covered by the Partitioned Normal Form (PNF).

Notes and references

[1] Codd, E.F. "Further Normalization of the Data Base Relational Model". (Presented at Courant Computer Science Symposia Series 6, "Data Base Systems", New York City, May 24€25, 1971.) IBM Research Report RJ909 (August 31, 1971). Republished in Randall J. Rustin (ed.), *Data Base Systems: Courant Computer Science Symposia Series 6*. Prentice-Hall, 1972.

[2] Codd, E. F. "Recent Investigations into Relational Data Base Systems". IBM Research Report RJ1385 (April 23, 1974). Republished in *Proc. 1974 Congress* (Stockholm, Sweden, 1974). , N.Y.: North-Holland (1974).

[3] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley (1999), p. 290

[4] Chris Date, for example, writes: "I believe firmly that anything less than a fully normalized design is *strongly contraindicated* ... [Y]ou should *"denormalize" only as a last resort*. That is, you should back off from a fully normalized design only if all other strategies for improving performance have somehow failed to meet requirements." Date, C.J. *Database in Depth: Relational Theory for Practitioners*. O'Reilly (2005), p. 152.

[5] "The adoption of a relational model of data ... permits the development of a universal data sub-language based on an applied predicate calculus. A first-order predicate calculus suffices if the collection of relations is in first normal form. Such a language would provide a yardstick of linguistic power for all other proposed data languages, and would itself be a strong candidate for embedding (with appropriate syntactic modification) in a variety of host languages (programming, command- or problem-oriented)." Codd, "A Relational Model of Data for Large Shared Data Banks" (http://www.acm.org/classics/nov95/toc.html), p. 381

[6] Codd, E.F. Chapter 23, "Serious Flaws in SQL", in *The Relational Model for Database Management: Version 2*. Addison-Wesley (1990), pp. 371€389

[7] Codd, E.F. "Further Normalization of the Data Base Relational Model", p. 34

[8] Date, C. J. "What First Normal Form Really Means" in *Date on Database: Writings 2000€2006* (Springer-Verlag, 2006), pp. 127€128.

[9] Zaniolo, Carlo. "A New Normal Form for the Design of Relational Database Schemata." *ACM Transactions on Database Systems* 7(3), September 1982.

[10] Codd, E. F. "Recent Investigations into Relational Data Base Systems". IBM Research Report RJ1385 (April 23, 1974). Republished in *Proc. 1974 Congress* (Stockholm, Sweden, 1974). New York, N.Y.: North-Holland (1974).

[11] Ronald Fagin. "Normal Forms and Relational Database Operators". ACM SIGMOD International Conference on Management of Data, May 31-June 1, 1979, Boston, Mass. Also IBM Research Report RJ2471, Feb. 1979.

[12] Ronald Fagin (1981) *A Normal Form for Relational Databases That Is Based on Domains and Keys* (http://www.almaden.ibm.com/cs/people/fagin/tods81.pdf), *Communications of the ACM*, vol. 6, pp. 387€415

[13] C.J. Date, Hugh Darwen, Nikos Lorentzos. *Temporal Data and the Relational Model*. Morgan Kaufmann (2002), p. 176

€ Paper: "Non First Normal Form Relations" by G. Jaeschke, H. -J Schek ; IBM Heidelberg Scientific Center. -> Paper studying normalization and denormalization operators nest and unnest as mildly described at the end of this wiki page.

Further reading

€ Litt's Tips: Normalization (http://www.troubleshooters.com/littstip/ltnorm.html)

€ Date, C. J. (1999), *An Introduction to Database Systems* (http://www.aw-bc.com/catalog/academic/product/0,1144,0321197844,00.html) (8th ed.). Addison-Wesley Longman. ISBN 0-321-19784-4.

€ Kent, W. (1983) *A Simple Guide to Five Normal Forms in Relational Database Theory* (http://www.bkent.net/Doc/simple5.htm), Communications of the ACM, vol. 26, pp.ƒ120€125

€ H.-J. Schek, P. Pistor Data Structures for an Integrated Data Base Management and Information Retrieval System

External links

- Database Normalization Basics (http://databases.about.com/od/specificproducts/a/normalization.htm) by Mike Chapple (About.com)
- Database Normalization Intro (http://www.databasejournal.com/sqletc/article.php/1428511), Part 2 (http://www.databasejournal.com/sqletc/article.php/26861_1474411_1)
- An Introduction to Database Normalization (http://mikehillyer.com/articles/an-introduction-to-database-normalization/) by Mike Hillyer.
- A tutorial on the first 3 normal forms (http://phlonx.com/resources/nf3/) by Fred Coulson
- DB Normalization Examples (http://www.dbnormalization.com/)
- Description of the database normalization basics (http://support.microsoft.com/kb/283878) by Microsoft
- Database Normalization and Design Techniques (http://www.barrywise.com/2008/01/database-normalization-and-design-techniques/) by Barry Wise, recommended reading for the Harvard MIS.
- A Simple Guide to Five Normal Forms in Relational Database Theory (http://www.bkent.net/Doc/simple5.htm)

Database storage structures

Database tables and indexes may be stored on disk in one of a number of forms, including ordered/unordered flat files, ISAM, heap files, hash buckets, or B+ trees. Each form has its own particular advantages and disadvantages. The most commonly used forms are B+ trees and ISAM. Such forms or structures are one aspect of the overall schema used by a database engine to store information.

Unordered

Unordered storage typically stores the records in the order they are inserted. Such storage offers good insertion efficiency ($O(1)$), but inefficient retrieval times ($O(n)$). Typically these retrieval times are better, however, as most databases use indexes on the primary keys, resulting in retrieval times of $O(\log n)$or $O(1)$for keys that are the same as the database row offsets within the storage system.

Ordered

Ordered storage typically stores the records in order and may have to rearrange or increase the file size when a new record is inserted, resulting in lower insertion efficiency. However, ordered storage provides more efficient retrieval as the records are pre-sorted, resulting in a complexity of $O(\log n)$.

Structured files

Heap files

- Simplest and most basic method
  - insert efficient, with new records added at the end of the file, providing chronological order
  - retrieval inefficient as searching has to be linear
  - deletion is accomplished by marking selected records as "deleted"
  - requires periodic reorganization if file is very volatile[*clarify*]
- Advantages
  - efficient for bulk loading data
  - efficient for relatively small relations as indexing overheads are avoided

- € efficient when retrievals involve large proportion of stored records
- € Disadvantages
  - € not efficient for selective retrieval using key values, especially if large
  - € sorting may be time-consuming
  - € not suitable for volatile[*clarify*] tables

Heap files are lists of unordered records of variable size. Although sharing a similar name, heap files are widely different from in-memory heaps.Wikipedia:Please clarify

Hash buckets

- € Hash functions calculate the address of the page in which the record is to be stored based on one or more fields in the record
  - € hashing functions chosen to ensure that addresses are spread evenly across the address space
  - € †occupancy, is generally 40% to 60% of the total file size
  - € unique address not guaranteed so collision detection and collision resolution mechanisms are required
- € Open addressing
- € Chained/unchained overflow
- € Pros and cons
  - € efficient for exact matches on key field
  - € not suitable for range retrieval, which requires sequential storage
  - € calculates where the record is stored based on fields in the record
  - € hash functions ensure even spread of data
  - € collisions are possible, so collision detection and restoration is required

B+ trees

These are the most commonly used in practice.

- € Time taken to access any record is the same because the same number of nodes is searched
- € Index is a full index so data file does not have to be ordered
- € Pros and cons
  - € versatile data structure € sequential as well as random access
  - € access is fast
  - € supports exact, range, part key and pattern matches efficiently
  - € volatile files are handled efficiently because index is dynamic € expands and contracts as table grows and shrinks
  - € less well suited to relatively stable files € in this case, ISAM is more efficient

Data orientation

Most conventional relational databases use "row-oriented" storage, meaning that all data associated with a given row is stored together. By contrast, column-oriented DBMS store all data from a given column together in order to more quickly serve data warehouse-style queries. Correlation databases are similar to row-based databases, but apply a layer of indirection to map multiple instances of the same value to the same numerical identifier.

Distributed database

A **distributed database** is a database in which storage devices are not all attached to a common processing unit such as the CPU, controlled by a **distributed database management system** (together sometimes called a **distributed database system**). It may be stored in multiple computers, located in the same physical location; or may be dispersed over a network of interconnected computers. Unlike parallel systems, in which the processors are tightly coupled and constitute a single database system, a distributed database system consists of loosely-coupled sites that share no physical components.

System administrators can distribute collections of data (e.g. in a database) across multiple physical locations. A distributed database can reside on network servers on the Internet, on corporate intranets or extranets, or on other company networks. Because they store data across multiple computers, distributed databases can improve performance at end-user worksites by allowing transactions to be processed on many machines, instead of being limited to one.[1]

Two processes ensure that the distributed databases remain up-to-date and current: replication and duplication.

1. Replication involves using specialized software that looks for changes in the distributive database. Once the changes have been identified, the replication process makes all the databases look the same. The replication process can be complex and time-consuming depending on the size and number of the distributed databases. This process can also require a lot of time and computer resources.
2. Duplication, on the other hand, has less complexity. It basically identifies one database as a master and then duplicates that database. The duplication process is normally done at a set time after hours. This is to ensure that each distributed location has the same data. In the duplication process, users may change only the master database. This ensures that local data will not be overwritten.

Both replication and duplication can keep the data current in all distributive locations.

Besides distributed database replication and fragmentation, there are many other distributed database design technologies. For example, local autonomy, synchronous and asynchronous distributed database technologies. These technologies' implementation can and does depend on the needs of the business and the sensitivity/confidentiality of the data stored in the database, and hence the price the business is willing to spend on ensuring data security, consistency and integrity.

When discussing access to distributed databases, Microsoft favors the term **distributed query**, which it defines in protocol-specific manner as "[a]ny SELECT, INSERT, UPDATE, or DELETE statement that references tables and rowsets from one or more external OLE DB data sources". Oracle provides a more language-centric view in which distributed queries and distributed transactions form part of **distributed SQL**.

Architecture

A database user accesses the distributed database through:

Local applications

: applications which do not require data from other sites.

Global applications

: applications which do require data from other sites.

A **homogeneous distributed database** has identical software and hardware running all databases instances, and may appear through a single interface as if it were a single database. A **heterogeneous distributed database** may have different hardware, operating systems, database management systems, and even data models for different databases.

Homogeneous DDBMS

In a homogeneous distributed database all sites have identical software and are aware of each other and agree to cooperate in processing user requests. Each site surrenders part of its autonomy in terms of right to change schema or software. A homogeneous DDBMS appears to the user as a single system. The homogeneous system is much easier to design and manage. The following conditions must be satisfied for homogeneous database:

- The operating system used, at each location must be same or compatible.Wikipedia:Avoid weasel wordsWikipedia:Please clarify
- The data structures used at each location must be same or compatible.
- The database application (or DBMS) used at each location must be same or compatible.

Heterogeneous DDBMS

In a heterogeneous distributed database, different sites may use different schema and software. Difference in schema is a major problem for query processing and transaction processing. Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing. In heterogeneous systems, different nodes may have different hardware & software and data structures at various nodes or locations are also incompatible. Different computers and operating systems, database applications or data models may be used at each of the locations. For example, one location may have the latest relational database management technology, while another location may store data using conventional files or old version of database management system. Similarly, one location may have the Windows NT operating system, while another may have UNIX. Heterogeneous systems are usually used when individual sites use their own hardware and software. On heterogeneous system, translations are required to allow communication between different sites (or DBMS). In this system, the users must be able to make requests in a database language at their local sites. Usually the SQL database language is used for this purpose. If the hardware is different, then the translation is straightforward, in which computer codes and word-length is changed. The heterogeneous system is often not technically or economically feasible. In this system, a user at one location may be able to read but not update the data at another location.

Important considerations

Care with a distributed database must be taken to ensure the following:

- The distribution is transparent *—* users must be able to interact with the system as if it were one logical system. This applies to the system's performance, and methods of access among other things.
- Transactions are transparent *—* each transaction must maintain database integrity across multiple databases. Transactions must also be divided into sub-transactions, each sub-transaction affecting one database system.

There are two principal approaches to store a relation r in a distributed database system:

A) Replication

B) Fragmentation/Partitioning

A) Replication: In replication, the system maintains several identical replicas of the same relation r in different sites.

- Data is more available in this scheme.
- Parallelism is increased when read request is served.
- Increases overhead on update operations as each site containing the replica needed to be updated in order to maintain consistency.
- Multi-datacenter replication provides geographical diversity: http://basho.com/tag/multi-datacenter-replication/

B) Fragmentation: The relation r is fragmented into several relations r_1, r_2, r_3....r_n in such a way that the actual relation could be reconstructed from the fragments and then the fragments are scattered to different locations. There are basically two schemes of fragmentation:

- Horizontal fragmentation - splits the relation by assigning each tuple of r to one or more fragments.
- Vertical fragmentation - splits the relation by decomposing the schema R of relation r.

Advantages

- Management of distributed data with different levels of transparency like network transparency, fragmentation transparency, replication transparency, etc.
- Increase reliability and availability
- Easier expansion
- Reflects organizational structure — database fragments potentially stored within the departments they relate to
- Local autonomy or site autonomy — a department can control the data about them (as they are the ones familiar with it)
- Protection of valuable data — if there were ever a catastrophic event such as a fire, all of the data would not be in one place, but distributed in multiple locations
- Improved performance — data is located near the site of greatest demand, and the database systems themselves are parallelized, allowing load on the databases to be balanced among servers. (A high load on one module of the database won't affect other modules of the database in a distributed database)
- Economics — it may cost less to create a network of smaller computers with the power of a single large computer
- Modularity — systems can be modified, added and removed from the distributed database without affecting other modules (systems)
- Reliable transactions - due to replication of the database
- Hardware, operating-system, network, fragmentation, DBMS, replication and location independence
- Continuous operation, even if some nodes go offline (depending on design)
- Distributed query processing can improve performance
- Distributed transaction management
- Single-site failure does not affect performance of system.
- All transactions follow A.C.I.D. property:
  - A-atomicity, the transaction takes place as a whole or not at all
  - C-consistency, maps one consistent DB state to another
  - I-isolation, each transaction sees a consistent DB
  - D-durability, the results of a transaction must survive system failures

The Merge Replication Method is popularly used to consolidate the data between databases.[*citation needed*]

Disadvantages

- Complexity — DBAs may have to do extra work to ensure that the distributed nature of the system is transparent. Extra work must also be done to maintain multiple disparate systems, instead of one big one. Extra database design work must also be done to account for the disconnected nature of the database — for example, joins become prohibitively expensive when performed across multiple systems.
- Economics — increased complexity and a more extensive infrastructure means extra labour costs
- Security — remote database fragments must be secured, and they are not centralized so the remote sites must be secured as well. The infrastructure must also be secured (for example, by encrypting the network links between remote sites).
- Difficult to maintain integrity — but in a distributed database, enforcing integrity over a network may require too much of the network's resources to be feasible
- Inexperience — distributed databases are difficult to work with, and in such a young field there is not much readily available experience in "proper" practice

- Lack of standards – there are no tools or methodologies yet to help users convert a centralized DBMS into a distributed DBMS[*citation needed*]
- Database design more complex – besides of the normal difficulties, the design of a distributed database has to consider fragmentation of data, allocation of fragments to specific sites and data replication
- Additional software is required
- Operating system should support distributed environment
- Concurrency control poses a major issue. It can be solved by locking and timestamping.
- Distributed access to data
- Analysis of distributed data

References

[1] O'Brien, J. & Marakas, G.M.(2008) Management Information Systems (pp. 185-189). New York, NY: McGraw-Hill Irwin

- M. T. , zsu and P. Valduriez, *Principles of Distributed Databases* (3rd edition) (2011), Springer, ISBN 978-1-4419-8833-1
- Elmasri and Navathe, *Fundamentals of database systems* (3rd edition), Addison-Wesley Longman, ISBN 0-201-54263-3
- *Oracle Database Administrator's Guide 10g* (Release 1), http://docs.oracle.com/cd/B14117_01/server.101/b10739/ds_concepts.htm

Federated database system

A **federated database system** is a type of meta-database management system (DBMS), which transparently maps multiple autonomous database systems into a single **federated database**. The constituent databases are interconnected via a computer network and may be geographically decentralized. Since the constituent database systems remain autonomous, a federated database system is a contrastable alternative to the (sometimes daunting) task of merging several disparate databases. A federated database, or **virtual database**, is a composite of all constituent databases in a federated database system. There is no actual data integration in the constituent disparate databases as a result of data federation.

McLeod and Heimbigner were among the first to define a federated database system, as one which "define[s] the architecture and interconnect[s] databases that minimize central authority yet support partial sharing and coordination among database systems".

Through data abstraction, federated database systems can provide a uniform user interface, enabling users and clients to store and retrieve data in multiple noncontiguous databases with a single query -- even if the constituent databases are heterogeneous. To this end, a federated database system must be able to decompose the query into subqueries for submission to the relevant constituent DBMS's, after which the system must composite the result sets of the subqueries. Because various database management systems employ different query languages, federated database systems can apply wrappers to the subqueries to translate them into the appropriate query languages.

- Note: this description of federated databases does not accurately reflect the McLeod/Heimbigner definition of a federated database. Rather, this description fits what McLeod/Heimbinger called a *composite* database. McLeod/Heimbigner's federated database is a collection of autonomous components that make their data available to other members of the federation through the publication of an export schema and access operations; there is no unified, central schema that encompasses the information available from the members of the federation.

Among other surveys, defines a Federated Database as a collection of cooperating component systems which are autonomous and are possibly heterogeneous. The three important components of an FDBS as pointed out in are

autonomy, heterogeneity and distribution. Another dimension which has also been considered is the Networking Environment Computer Network, e.g., many DBSs over a LAN or many DBSs over a WAN update related functions of participating DBSs (e.g., no updates, nonatomic transitions, atomic updates).

FDBS architecture

A DBMS can be classified as either centralized or distributed. A centralized system manages a single database while distributed manages multiple databases. A component DBS in a DBMS may be centralized or distributed. A multiple DBS (MDBS) can be classified into two types depending on the autonomy of the component DBS as federated and non federated. A nonfederated database system is an integration of component DBMS that are not autonomous. A federated database system consists of component DBS that are autonomous yet participate in a federation to allow partial and controlled sharing of their data.

Federated architectures differ based on levels of integration with the component database systems and the extent of services offered by the federation. A FDBS can be categorized as loosely or tightly coupled systems.

- Loosely Coupled require component databases to construct their own federated schema. A user will typically access other component database systems by using a multidatabase language but this removes any levels of location transparency, forcing the user to have direct knowledge of the federated schema. A user imports the data they require from other component databases and integrates it with their own to form a federated schema.
- Tightly coupled system consists of component systems that use independent processes to construct and publicize an integrated federated schema.

Multiple DBS of which FDBS are a specific type can be characterized along three dimensions: Distribution, Heterogeneity and Autonomy. Another characterization could be based on the dimension of networking, for example single databases or multiple databases in a LAN or WAN.

Distribution

Distribution of data in an FDBS is due to the existence of a multiple DBS before an FDBS is built. Data can be distributed among multiple DB which could be stored in a single computer or multiple computers. These computers could be geographically located in different places but interconnected by a network. The benefits of data distribution help in increased availability and reliability as well as improved access times.

Heterogeneity

Heterogeneities in databases arise due to factors such as differences in structures, semantics of data, the constraints supported or query language. Differences in structure occur when two data models provide different primitives such as object oriented (OO) models that support specialization and inheritance and relational models that do not. Differences due to constraints occur when two models support two different constraints. For example the set type in CODASYL schema may be partially modeled as a referential integrity constraint in a relationship schema. CODASYL supports insertion and retention that are not captured by referential integrity alone. The query language supported by one DBMS can also contribute to heterogeneity between other component DBMSs. For example, differences in query languages with the same data models or different versions of query languages could contribute to heterogeneity.

Semantic heterogeneities arise when there is a disagreement about meaning, interpretation or intended use of data. At the schema and data level, classification of possible heterogeneities include:

- Naming conflicts e.g. databases using different names to represent the same concept.
- Domain conflicts or data representation conflicts e.g. databases using different values to represent same concept.
- Precision conflicts e.g. databases using same data values from domains of different cardinalities for same data.
- Metadata conflicts e.g. same concepts are represented at schema level and instance level.

- Data conflicts e.g. missing attributes
- Schema conflicts e.g. table versus table conflict which includes naming conflicts, data conflicts etc.

In creating a federated schema, one has to resolve such heterogeneities before integrating the component DB schemas.

Schema matching, schema mapping

Dealing with incompatible data types or query syntax is not the only obstacle to a concrete implementation of an FDBS. In systems that are not planned top-down, a generic problem lies in matching semantically equivalent, but differently named parts from different schemas (=data models) (tables, attributes). A pairwise mapping between *n* attributes would result in $\frac{n(n-1)}{2}$ mapping rules (given equivalence mappings) - a number that quickly gets too large for practical purposes. A common way out is to provide a global schema that comprises the relevant parts of all member schemas and provide mappings in the form of database views. Two principal solutions can be realized, depending on the direction of the mapping:

1. Global as View (GaV): the global schema is defined in terms of the underlying schemas
2. Local as View (LaV): the local schemas are defined in terms of the global schema

Both are explained in more detail in the article Data integration. Alternate approaches to the schema matching problem and a classification of the same are explained in more detail in the article Schema Matching

Autonomy

Fundamental to the difference between an MDBS and an FDBS is the concept of autonomy. It is important to understand the aspects of autonomy for component databases and how they can be addressed when a component DBS participates in an FDBS. There are four kinds of autonomies addressed:

- Design Autonomy which refers to ability to choose its design irrespective of data, query language or conceptualization, functionality of the system implementation.

Heterogeneities in an FDBS are primarily due to design autonomy.

- Communication autonomy refers to the general operation of the DBMS to communicate with other DBMS or not.
- Execution autonomy allows a component DBMS to control the operations requested by local and external operations.
- Association autonomy gives a power to component DBS to disassociate itself from a federation which means FDBS can operate independently of any single DBS.

The ANSI/X3/SPARC Study Group outlined a three level data description architecture, the components of which are the conceptual schema, internal schema and external schema of databases. The three level architecture is however inadequate to describing the architectures of an FDBS. It was therefore extended to support the three dimensions of the FDBS namely Distribution, Autonomy and Heterogeneity. The five level schema architecture is explained below.

Concurrency control

The *Heterogeneity* and *Autonomy* requirements pose special challenges concerning concurrency control in an FDBS, which is crucial for the correct execution of its concurrent transactions (see also Global concurrency control). Achieving global serializability, the major correctness criterion, under these requirements has been characterized as very difficult and unsolved. Commitment ordering, introduced in 1991, has provided a general solution for this issue (See Global serializability; See Commitment ordering also for the architectural aspects of the solution).

Five Level Schema Architecture for FDBSs

The five level schema architecture includes the following:

- € Local Schema is the conceptual concept expressed in primary data model of component DBMS.
- € Component Schema is derived by translating local schema into a model called the canonical data model or common data model. They are useful when semantics missed in local schema are incorporated in the component. They help in integration of data for tightly coupled FDBS.
- € Export Schema represents a subset of a component schema that is available to the FDBS. It may include access control information regarding its use by specific federation user. The export schema help in managing flow of control of data.
- € Federated Schema is an integration of multiple export schema. It includes information on data distribution that is generated when integrating export schemas.
- € External Schema defines a schema for a user/applications or a class of users/applications.

While accurately representing the state of the art in data integration, the Five Level Schema Architecture above does suffer from a major drawback, namely IT imposed look and feel. Modern data users demand control over how data is presented; their needs are somewhat in conflict with such bottom-up approaches to data integration.

References

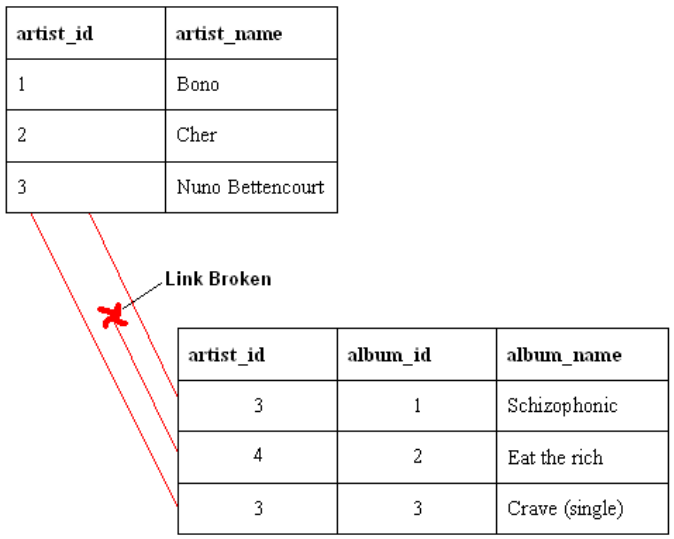
External links

- € Schema coordination in federated database management: a comparison with schema integration (http://citeseer.ist.psu.edu/cache/papers/cs/9149/http:zSzzSzwww.bm.ust.hkzSz~zhaozSzDSS96.pdf/schema-coordination-in-federated.pdf)
- € Storage of Behaviour of Object Database (http://www.computing.dcu.ie/~dalenk/publications/PhD Transfer talk.ppt)
- € DB2 and Federated Databases (http://www.ibm.com/developerworks/db2/library/techarticle/dm-0504zikopoulos/)
- € Tutorial on Federated Database (http://www.vldb.org/conf/1991/P489.PDF)
- € GaV and LaV explained (http://www.dcs.bbk.ac.uk/~lucas/talks/SCSIS_RD_200507.pps)
- € Issues of where to perform the join aka "pushdown" and other performance characteristics (http://www.ibm.com/developerworks/db2/library/techarticle/0304lurie/0304lurie.html)
- € Worked example federating Oracle, Informix, DB2, and Excel (http://www.ibm.com/developerworks/db2/library/techarticle/0307lurie/0307lurie.html)
- € Composite Information Server - a commercial federated database product (http://www.compositesw.com/products/cis.shtml)
- € Freitas, Andr…, Edward Curry, Jo†o Gabriel Oliveira, and Sean O,Riain. 2012. •Querying Heterogeneous Datasets on the Linked Data Web: Challenges, Approaches, and Trends.‡ (http://www.edwardcurry.org/publications/freitas_IC_12.pdf) IEEE Internet Computing 16 (1): 24€33.
- € IBM Gaian Database: A dynamic Distributed Federated Database (https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=f6ce657b-f385-43b2-8350-458e6e4a344f)
- € Federated system and methods and mechanisms of implementing and using such a system (http://www.google.com/patents/US7392255)

Referential integrity

Referential integrity is a property of data which, when satisfied, requires every value of one attribute (column) of a relation (table) to exist as a value of another attribute in a different (or the same) relation (table).

For referential integrity to hold in a relational database, any field in a table that is declared a foreign key can contain either a null value, or only values from a parent table's primary key or a candidate key.[1] In other words, when a foreign key value is used it must reference a valid, existing primary key in the parent table. For instance, deleting a record that contains a value referred to by a foreign key in another table would break referential integrity. Some relational database management systems (RDBMS) can enforce referential integrity, normally either by deleting the foreign key rows as well to maintain integrity, or by returning an error and not performing the delete. Which method is used may be determined by a referential integrity constraint defined in a data dictionary.



An example of a database that has not enforced **referential integrity**. In this example, there is a foreign key (artist_id) value in the album table that references a non-existent artist *f* in other words there is a foreign key value with no corresponding primary key value in the referenced table. What happened here was that there was an artist called "Aerosmith", with an artist_id of 4, which was deleted from the artist table. However, the album "Eat the Rich" referred to this artist. With referential integrity enforced, this would not have been possible.

Formalization

An **inclusion dependency** over two (possibly identical) predicates R and S from a schema is written $R[A_1, ..., A_n] \subseteq S[B_1, ..., B_n]$, where the A_i, B_i are distinct attributes (column names) of R and S. It implies that the tuples of values appearing in columns $A_1, ..., A_n$ for facts of R must also appear as a tuple of values in columns $B_1, ..., B_n$ for some fact of S.

Logical implication between inclusion dependencies can be axiomatized by inference rules [2] and can be decided by a PSPACE algorithm. The problem can be shown to be PSPACE-complete by reduction from the acceptance problem for a linear bounded automaton.[3] However, logical implication between dependencies that can be inclusion dependencies or functional dependencies is undecidable by reduction from the word problem for monoids.[4]

References

[1] Coronel et al. (2013). Database Systems 10th ed. Cengage Learning, ISBN 978-1-111-96960-8
[2] Abiteboul, Hull, Vianu. *Foundations of Databases* Addison-Wesley, 1994. Section 9.1, p. 193. Freely available online (http://webdam.inria.fr/Alice/).
[3] ibid., p. 196
[4] ibid., p. 199

Relational algebra

In computer science, **relational algebra** is an offshoot of first-order logic and of algebra of sets concerned with operations over finitary relations, usually made more convenient to work with by identifying the components of a tuple by a name (called attribute) rather than by a numeric column index, which is called a relation in database terminology.

The main application of relational algebra is providing a theoretical foundation for relational databases, particularly query languages for such databases, chief among which is SQL.

Introduction

Relational algebra received little attention outside of pure mathematics until the publication of E.F. Codd's relational model of data in 1970. Codd proposed such an algebra as a basis for database query languages. (See section Implementations.)

Both a named and an unnamed perspective are possible for relational algebra, depending on whether the tuples are endowed with component names or not. In the unnamed perspective, a tuple is simply a member of a Cartesian product. In the named perspective, tuples are functions from a finite set *U* of attributes (of the relation) to a domain of values (assumed distinct from *U*).[1] The relational algebras obtained from the two perspectives are equivalent.[2] The typical undergraduate textbooks present only the named perspective though, and this article follows suit.

Relational algebra is essentially equivalent in expressive power to relational calculus (and thus first-order logic); this result is known as Codd's theorem. One must be careful to avoid a mismatch that may arise between the two languages because negation, applied to a formula of the calculus, constructs a formula that may be true on an infinite set of possible tuples, while the difference operator of relational algebra always returns a finite result. To overcome these difficulties, Codd restricted the operands of relational algebra to finite relations only and also proposed restricted support for negation (NOT) and disjunction (OR). Analogous restrictions are found in many other logic-based computer languages. Codd defined the term ***relational completeness*** to refer to a language that is complete with respect to first-order predicate calculus apart from the restrictions he proposed. In practice the restrictions have no adverse effect on the applicability of his relational algebra for database purposes.

Primitive operations

As in any algebra, some operators are primitive and the others are derived in terms of the primitive ones. It is useful if the choice of primitive operators parallels the usual choice of primitive logical operators.

Five primitive operators of Codd's algebra are the *selection*, the *projection*, the *Cartesian product* (also called the *cross product* or *cross join*), the *set union*, and the *set difference*. Another operator, *rename* was not noted by Codd, but the need for it is shown by the inventors of ISBL. These six operators are fundamental in the sense that omitting any one of them causes a loss of expressive power. Many other operators have been defined in terms of these six. Among the most important are set intersection, division, and the natural join. In fact ISBL made a compelling case for replacing the Cartesian product with the natural join, of which the Cartesian product is a degenerate case.

Altogether, the operators of relational algebra have an expressive power identical to that of domain relational calculus or tuple relational calculus. However, for the reasons given in section Introduction, relational algebra is less expressive than first-order predicate calculus without function symbols. Relational algebra corresponds to a subset of first-order logic, namely Horn clauses without recursion and negation.

Set operators

The relational algebra uses set union, set difference, and Cartesian product from set theory, but adds additional constraints to these operators.

For set union and set difference, the two relations involved must be *union-compatible f* that is, the two relations must have the same set of attributes. Because set intersection can be defined in terms of set difference, the two relations involved in set intersection must also be union-compatible.

For the Cartesian product to be defined, the two relations involved must have disjoint headers *f* that is, they must not have a common attribute name.

In addition, the Cartesian product is defined differently from the one in set theory in the sense that tuples are considered to be "shallow" for the purposes of the operation. That is, the Cartesian product of a set of *n*-tuples with a set of *m*-tuples yields a set of "flattened" (*n*ƒ+ƒ*m*)-tuples (whereas basic set theory would have prescribed a set of 2-tuples, each containing an *n*-tuple and an *m*-tuple). More formally, *R* ‡ *S* is defined as follows:

$$R \ddagger S = \{(r_1, r_2, ..., r_n, s_1, s_2, ..., s_m) \mid (r_1, r_2, ..., r_n) \hat{} R, (s_1, s_2, ..., s_m) \hat{} S\}$$

The cardinality of the Cartesian product is the product of the cardinalities of its factors, i.e., $|R \ddagger S| = |R| \ddagger |S|$.

Projection (•)

A **projection** is a unary operation written as $\pi_{a_1,\ldots,a_n}(R)$ where $a_1, \ldots, a_n$ is a set of attribute names. The result of such projection is defined as the set that is obtained when all tuples in *R* are restricted to the set $\{a_1, \ldots, a_n\}$. This specifies the specific subset of columns (attributes of each tuple) to be retrieved. To obtain the names and phone numbers from an address book, the projection might be written $\pi_{\mathrm{contactName},\ \mathrm{contactPhoneNumber}}(\mathrm{addressBook})$. The result of that projection would be a relation which contains only the contactName and contactPhoneNumber attributes for each unique entry in addressBook.

Selection (€)

A **generalized selection** is a unary operation written as $\sigma_{\varphi}(R)$ where φ is a propositional formula that consists of atoms as allowed in the normal selection and the logical operators $\wedge$ (and), $\vee$ (or) and $\neg$ (negation). This selection selects all those tuples in *R* for which φ holds.

To obtain a listing of all friends or business associates in an address book, the selection might be written as $\sigma_{\mathrm{isFriend} = \mathrm{true} \vee \mathrm{isBusinessContact} = \mathrm{true}}(\mathrm{addressBook})$. The result would be a relation containing every attribute of every unique record where isFriend is true or where isBusinessContact is true.

In Codd's 1970 paper, selection is called restriction.

Rename (•)

A **rename** is a unary operation written as $\rho_{a/b}(R)$ where the result is identical to *R* except that the *b* attribute in all tuples is renamed to an *a* attribute. This is simply used to rename the attribute of a relation or the relation itself.
To rename the 'isFriend' attribute to 'isBusinessContact' in a relation, $\rho_{\text{isBusinessContact / isFriend}}(\text{addressBook})$ might be used.

Joins and join-like operators

Natural join (€)

Natural join (⋈) is a binary operator that is written as (*R* ⋈ *S*) where *R* and *S* are relations.[3] The result of the natural join is the set of all combinations of tuples in *R* and *S* that are equal on their common attribute names. For an example consider the tables *Employee* and *Dept* and their natural join:

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales

DeptName	Manager
Finance	George
Sales	Harriet
Production	Charles

Name	EmpId	DeptName	Manager
Harry	3415	Finance	George
Sally	2241	Sales	Harriet
George	3401	Finance	George
Harriet	2202	Sales	Harriet

This can also be used to define composition of relations. For example, the composition of *Employee* and *Dept* is their join as shown above, projected on all but the common attribute *DeptName*. In category theory, the join is precisely the fiber product.

The natural join is arguably one of the most important operators since it is the relational counterpart of logical AND. Note carefully that if the same variable appears in each of two predicates that are connected by AND, then that variable stands for the same thing and both appearances must always be substituted by the same value. In particular, natural join allows the combination of relations that are associated by a foreign key. For example, in the above example a foreign key probably holds from *Employee.DeptName* to *Dept.DeptName* and then the natural join of *Employee* and *Dept* combines all employees with their departments. Note that this works because the foreign key holds between attributes with the same name. If this is not the case such as in the foreign key from *Dept.manager* to *Employee.Name* then we have to rename these columns before we take the natural join. Such a join is sometimes also referred to as an **equijoin** (see €-join).

More formally the semantics of the natural join are defined as follows:

$$R \bowtie S = \{t \cup s \mid t \in R \ \wedge \ s \in S \ \wedge \ Fun(t \cup s)\}$$

where *Fun* is a predicate that is true for a relation *r* if and only if *r* is a function. It is usually required that *R* and *S* must have at least one common attribute, but if this constraint is omitted, and *R* and *S* have no common attributes, then the natural join becomes exactly the Cartesian product.

The natural join can be simulated with Codd's primitives as follows. Assume that $c_1,\ldots,c_m$ are the attribute names common to *R* and *S*, $r_1,\ldots,r_n$ are the attribute names unique to *R* and $s_1,\ldots,s_k$ are the attribute unique to *S*. Furthermore assume that the attribute names $x_1,\ldots,x_m$ are neither in *R* nor in *S*. In a first step we can now rename the common attribute names in *S*:

$$T = \rho_{x_1/c_1,\ldots,x_m/c_m}(S) = \rho_{x_1/c_1}(\rho_{x_2/c_2}(\ldots \rho_{x_m/c_m}(S)\ldots))$$

Then we take the Cartesian product and select the tuples that are to be joined:

$$P = \sigma_{c_1=x_1,\ldots,c_m=x_m}(R \times T) = \sigma_{c_1=x_1}(\sigma_{c_2=x_2}(\ldots \sigma_{c_m=x_m}(R \times T)\ldots))$$

Finally we take a projection to get rid of the renamed attributes:

$$U = \pi_{r_1,\ldots,r_n,c_1,\ldots,c_m,s_1,\ldots,s_k}(P)$$

,-join and equijoin

Consider tables *Car* and *Boat* which list models of cars and boats and their respective prices. Suppose a customer wants to buy a car and a boat, but she does not want to spend more money for the boat than for the car. The ˆ-join ($\bowtie_{\hat{}}$) on the relation *CarPrice* ‰ *BoatPrice* produces a table with all the possible options. When using a condition where the attributes are equal, for example Price, then the condition may be specified as *Price=Price* or alternatively *(Price)* itself.

CarModel	CarPrice
CarA	20,000
CarB	30,000
CarC	50,000

BoatModel	BoatPrice
Boat1	10,000
Boat2	40,000
Boat3	60,000

CarModel	CarPrice	BoatModel	BoatPrice
CarA	20,000	Boat1	10,000
CarB	30,000	Boat1	10,000
CarC	50,000	Boat1	10,000
CarC	50,000	Boat2	40,000

If we want to combine tuples from two relations where the combination condition is not simply the equality of shared attributes then it is convenient to have a more general form of join operator, which is the ˆ-join (or theta-join). The ˆ-join is a binary operator that is written as $R \underset{a\ \theta\ b}{\bowtie} S$ or $R \underset{a\ \theta\ v}{\bowtie} S$ where *a* and *b* are attribute names, ˆ is a binary relation in the set {<, Š, =, >, ‰}, *v* is a value constant, and *R* and *S* are relations. The result of this operation consists of all combinations of tuples in *R* and *S* that satisfy the relation ˆ. The result of the ˆ-join is defined only if the headers of *S* and *R* are disjoint, that is, do not contain a common attribute.

The simulation of this operation in the fundamental operations is therefore as follows:

$R \bowtie_{\hat{}} S = \text{‰}_{\hat{}}(R \ddagger S)$

In case the operator ˆ is the equality operator (=) then this join is also called an **equijoin**.

Note, however, that a computer language that supports the natural join and rename operators does not need ˆ-join as well, as this can be achieved by selection from the result of a natural join (which degenerates to Cartesian product when there are no shared attributes).

Semijoin (•)(,)

The left semijoin is joining similar to the natural join and written as $R \ltimes S$ where *R* and *S* are relations.[4] The result of this semijoin is the set of all tuples in *R* for which there is a tuple in *S* that is equal on their common attribute names. For an example consider the tables *Employee* and *Dept* and their semi join:

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Production

DeptName	Manager
Sales	Bob
Sales	Thomas
Production	Katie
Production	Mark

Name	EmpId	DeptName
Sally	2241	Sales
Harriet	2202	Production

More formally the semantics of the semijoin is defined as follows:

$R \ltimes S = \{\, t \in R, s \in S, \mathit{Fun}\,(t \cup s)\,\}$

where *Fun*(*r*) is as in the definition of natural join.

The semijoin can be simulated using the natural join as follows. If a_1, ..., a_n are the attribute names of *R*, then

$R \ltimes S = \pi_{a1,..,a_n}(R \bowtie S)$.

Since we can simulate the natural join with the basic operators it follows that this also holds for the semijoin.

Antijoin (,)

The antijoin, written as $R \triangleright S$ where *R* and *S* are relations, is similar to the semijoin, but the result of an antijoin is only those tuples in *R* for which there is *no* tuple in *S* that is equal on their common attribute names.[5]

For an example consider the tables *Employee* and *Dept* and their antijoin:

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Production

DeptName	Manager
Sales	Sally
Production	Harriet

Name	EmpId	DeptName
Harry	3415	Finance
George	3401	Finance

The antijoin is formally defined as follows:

$R \triangleright S = \{ t : t \in R \wedge \neg\exists\, s \in S : Fun\,(t \cup s) \}$

or

$R \triangleright S = \{ t : t \in R$, there is no tuple *s* of *S* that satisfies $Fun\,(t \cup s) \}$

where *Fun*(*r*) is as in the definition of natural join.

The antijoin can also be defined as the complement of the semijoin, as follows:

$R \triangleright S = R \setminus (R \ltimes S)$

Given this, the antijoin is sometimes called the anti-semijoin, and the antijoin operator is sometimes written as semijoin symbol with a bar above it, instead of $\triangleright$.

Division (*f*)

The division is a binary operation that is written as *R* Š *S*. The result consists of the restrictions of tuples in *R* to the attribute names unique to *R*, i.e., in the header of *R* but not in the header of *S*, for which it holds that all their combinations with tuples in *S* are present in *R*. For an example see the tables *Completed*, *DBProject* and their division:

Student	Task
Fred	Database1
Fred	Database2
Fred	Compiler1
Eugene	Database1
Eugene	Compiler1
Sarah	Database1
Sarah	Database2

Task
Database1
Database2

Student
Fred
Sarah

If *DBProject* contains all the tasks of the Database project, then the result of the division above contains exactly the students who have completed both of the tasks in the Database project.

More formally the semantics of the division is defined as follows:

$$R \div S = \{ t[a_1,...,a_n] : t \in R \wedge \forall s \in S ((t[a_1,...,a_n] \cup s) \in R) \}$$

where $\{a_1,...,a_n\}$ is the set of attribute names unique to R and $t[a_1,...,a_n]$ is the restriction of t to this set. It is usually required that the attribute names in the header of S are a subset of those of R because otherwise the result of the operation will always be empty.

The simulation of the division with the basic operations is as follows. We assume that $a_1,...,a_n$ are the attribute names unique to R and $b_1,...,b_m$ are the attribute names of S. In the first step we project R on its unique attribute names and construct all combinations with tuples in S:

$$T := \pi_{a_1,...,a_n}(R) \times S$$

In the prior example, T would represent a table such that every Student (because Student is the unique key / attribute of the Completed table) is combined with every given Task. So Eugene, for instance, would have two rows, Eugene -> Database1 and Eugene -> Database2 in T.

In the next step we subtract R from this relation:

$$U := T - R$$

Note that in U we have the possible combinations that "could have" been in R, but weren't. So if we now take the projection on the attribute names unique to R then we have the restrictions of the tuples in R for which not all combinations with tuples in S were present in R:

$$V := \pi_{a_1,...,a_n}(U)$$

So what remains to be done is take the projection of R on its unique attribute names and subtract those in V:

$$W := \pi_{a_1,...,a_n}(R) - V$$

Common extensions

In practice the classical relational algebra described above is extended with various operations such as outer joins, aggregate functions and even transitive closure.

Outer joins

Whereas the result of a join (or inner join) consists of tuples formed by combining matching tuples in the two operands, an outer join contains those tuples and additionally some tuples formed by extending an unmatched tuple in one of the operands by "fill" values for each of the attributes of the other operand. Note that outer joins are not considered part of the classical relational algebra discussed so far.

The operators defined in this section assume the existence of a *null* value, •, which we do not define, to be used for the fill values; in practice this corresponds to the NULL in SQL. In order to make subsequent selection operations on the resulting table meaningful, a semantic meaning needs to be assigned to nulls; in Codd's approach the propositional logic used by the selection is extended to a three-valued logic, although we elide those details in this article.

Three outer join operators are defined: left outer join, right outer join, and full outer join. (The word "outer" is sometimes omitted.)

Left outer join (*ƒ*)

The left outer join is written as *R* ‹ *S* where *R* and *S* are relations.[6] The result of the left outer join is the set of all combinations of tuples in *R* and *S* that are equal on their common attribute names, in addition (loosely speaking) to tuples in *R* that have no matching tuples in *S*.

For an example consider the tables *Employee* and *Dept* and their left outer join:

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

DeptName	Manager
Sales	Harriet
Production	Charles

Name	EmpId	DeptName	Manager
Harry	3415	Finance	Œ
Sally	2241	Sales	Harriet
George	3401	Finance	Œ
Harriet	2202	Sales	Harriet
Tim	1123	Executive	Œ

In the resulting relation, tuples in *S* which have no common values in common attribute names with tuples in *R* take a *null* value, •.

Since there are no tuples in *Dept* with a *DeptName* of *Finance* or *Executive*, •s occur in the resulting relation where tuples in *Employee* have a *DeptName* of *Finance* or *Executive*.

Let r_1, r_2, ..., r_n be the attributes of the relation *R* and let {(•, ..., •)} be the singleton relation on the attributes that are *unique* to the relation *S* (those that are not attributes of *R*). Then the left outer join can be described in terms of the natural join (and hence using basic operators) as follows:

$$(R \bowtie S) \cup ((R - \pi_{r_1, r_2, \ldots, r_n}(R \bowtie S)) \times \{(\omega, \ldots \omega)\})$$

Right outer join („)

The right outer join behaves almost identically to the left outer join, but the roles of the tables are switched.

The right outer join of relations *R* and *S* is written as *R* Œ *S*.[7] The result of the right outer join is the set of all combinations of tuples in *R* and *S* that are equal on their common attribute names, in addition to tuples in *S* that have no matching tuples in *R*.

For example consider the tables *Employee* and *Dept* and their right outer join:

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

DeptName	Manager
Sales	Harriet
Production	Charles

Name	EmpId	DeptName	Manager
Sally	2241	Sales	Harriet
Harriet	2202	Sales	Harriet
Œ	Œ	Production	Charles

In the resulting relation, tuples in *R* which have no common values in common attribute names with tuples in *S* take a *null* value, •.

Since there are no tuples in *Employee* with a *DeptName* of *Production*, •s occur in the Name attribute of the resulting relation where tuples in *DeptName* had tuples of *Production*.

Let s_1, s_2, ..., s_n be the attributes of the relation *S* and let {(•, ..., •)} be the singleton relation on the attributes that are *unique* to the relation *R* (those that are not attributes of *S*). Then, as with the left outer join, the right outer join

can be simulated using the natural join as follows:

$$(R \bowtie S) \cup (\{(\omega, \ldots, \omega)\} \times (S - \pi_{s_1, s_2, \ldots, s_n}(R \bowtie S)))$$

Full outer join (…)

The **outer join** or **full outer join** in effect combines the results of the left and right outer joins.

The full outer join is written as *R* • *S* where *R* and *S* are relations.[8] The result of the full outer join is the set of all combinations of tuples in *R* and *S* that are equal on their common attribute names, in addition to tuples in *S* that have no matching tuples in *R* and tuples in *R* that have no matching tuples in *S* in their common attribute names.

For an example consider the tables *Employee* and *Dept* and their full outer join:

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

DeptName	Manager
Sales	Harriet
Production	Charles

Name	EmpId	DeptName	Manager
Harry	3415	Finance	Œ
Sally	2241	Sales	Harriet
George	3401	Finance	Œ
Harriet	2202	Sales	Harriet
Tim	1123	Executive	Œ
Œ	Œ	Production	Charles

In the resulting relation, tuples in *R* which have no common values in common attribute names with tuples in *S* take a *null* value, •. Tuples in *S* which have no common values in common attribute names with tuples in *R* also take a *null* value, •.

The full outer join can be simulated using the left and right outer joins (and hence the natural join and set union) as follows:

R • *S* = (*R* ‹ *S*) ∪ (*R* Œ *S*)

Operations for domain computations

There is nothing in relational algebra introduced so far that would allow computations on the data domains (other than evaluation of propositional expressions involving equality). For example, it's not possible using only the algebra introduced so far to write an expression that would multiply the numbers from two columns, e.g. a unit price with a quantity to obtain a total price. Practical query languages have such facilities, e.g. the SQL SELECT allows arithmetic operations to define new columns in the result `SELECT unit_price * quantity AS total_price FROM t`, and a similar facility is provided more explicitly by Tutorial D's EXTEND keyword. In database theory, this is called **extended projection**.:213

Aggregation

Furthermore, computing various functions on a column, like the summing up its elements, is also not possible using the relational algebra introduced so far. There are five aggregate functions that are included with most relational database systems. These operations are Sum, Count, Average, Maximum and Minimum. In relational algebra the aggregation operation over a schema (A_1, A_2, ... A_n) is written as follows:

G_1, G_2, ..., G_m *g* $f1(A_1')$, $f_2(A_2')$, ..., $f_k(A_k')$ (**r**)

where each A_j', 1 Š j Š k, is one of the original attributes A_i, 1 Š i Š n.

The attributes preceding the *g* are grouping attributes, which function like a "group by" clause in SQL. Then there are an arbitrary number of aggregation functions applied to individual attributes. The operation is applied to an arbitrary relation **r**. The grouping attributes are optional, and if they are not supplied, the aggregation functions are

applied across the entire relation to which the operation is applied.

Let's assume that we have a table named Account with three columns, namely Account_Number, Branch_Name and Balance. We wish to find the maximum balance of each branch. This is accomplished by $_{\text{Branch_Name}}\mathcal{G}_{\text{Max(Balance)}}$(Account). To find the highest balance of all accounts regardless of branch, we could simply write $\mathcal{G}_{\text{Max(Balance)}}$(Account).

Transitive closure

Although relational algebra seems powerful enough for most practical purposes, there are some simple and natural operators on relations which cannot be expressed by relational algebra. One of them is the transitive closure of a binary relation. Given a domain *D*, let binary relation *R* be a subset of *D*‡*D*. The transitive closure R^+ of *R* is the smallest subset of *D*‡*D* containing *R* which satisfies the following condition:

$$\forall x \forall y \forall z \left((x, y) \in R^+ \wedge (y, z) \in R^+ \Rightarrow (x, z) \in R^+ \right)$$

There is no relational algebra expression *E*(*R*) taking *R* as a variable argument which produces R^+. This can be proved using the fact that, given a relational expression *E* for which it is claimed that *E*(*R*) = R^+, where *R* is a variable, we can always find an instance ***r*** of *R* (and a corresponding domain **d**) such that *E*(***r***) Ž ***r***$^+$.

SQL however officially supports such fixpoint queries since 1999, and it had vendor-specific extensions in this direction well before that.

Use of algebraic properties for query optimization

Queries can be represented as a tree, where

- the internal nodes are operators,
- leaves are relations,
- subtrees are subexpressions.

Our primary goal is to transform expression trees into equivalent expression trees, where the average size of the relations yielded by subexpressions in the tree is smaller than it was before the optimization. Our secondary goal is to try to form common subexpressions within a single query, or if there is more than one query being evaluated at the same time, in all of those queries. The rationale behind the second goal is that it is enough to compute common subexpressions once, and the results can be used in all queries that contain that subexpression.

Here we present a set of rules that can be used in such transformations.

Selection

Rules about selection operators play the most important role in query optimization. Selection is an operator that very effectively decreases the number of rows in its operand, so if we manage to move the selections in an expression tree towards the leaves, the internal relations (yielded by subexpressions) will likely shrink.

Basic selection properties

Selection is idempotent (multiple applications of the same selection have no additional effect beyond the first one), and commutative (the order selections are applied in has no effect on the eventual result).

1. $\sigma_A(R) = \sigma_A\sigma_A(R)$
2. $\sigma_A\sigma_B(R) = \sigma_B\sigma_A(R)$

Breaking up selections with complex conditions

A selection whose condition is a conjunction of simpler conditions is equivalent to a sequence of selections with those same individual conditions, and selection whose condition is a disjunction is equivalent to a union of selections. These identities can be used to merge selections so that fewer selections need to be evaluated, or to split them so that the component selections may be moved or optimized separately.

1. $\sigma_{A \wedge B}(R) = \sigma_A(\sigma_B(R)) = \sigma_B(\sigma_A(R))$
2. $\sigma_{A \vee B}(R) = \sigma_A(R) \cup \sigma_B(R)$

Selection and cross product

Cross product is the costliest operator to evaluate. If the input relations have *N* and *M* rows, the result will contain NM rows. Therefore it is very important to do our best to decrease the size of both operands before applying the cross product operator.

This can be effectively done, if the cross product is followed by a selection operator, e.g. σ_A (*R* ‡ *P*). Considering the definition of join, this is the most likely case. If the cross product is not followed by a selection operator, we can try to push down a selection from higher levels of the expression tree using the other selection rules.

In the above case we break up condition *A* into conditions *B*, *C* and *D* using the split rules about complex selection conditions, so that *A* = *B* ∧ *C* ∧ *D* and *B* only contains attributes from *R*, *C* contains attributes only from *P* and *D* contains the part of *A* that contains attributes from both *R* and *P*. Note, that *B*, *C* or *D* are possibly empty. Then the following holds:

$$\sigma_A(R \times P) = \sigma_{B \wedge C \wedge D}(R \times P) = \sigma_D(\sigma_B(R) \times \sigma_C(P))$$

Selection and set operators

Selection is distributive over the setminus, intersection, and union operators. The following three rules are used to push selection below set operations in the expression tree. Note, that in the setminus and the intersection operators it is possible to apply the selection operator to only one of the operands after the transformation. This can make sense in cases, where one of the operands is small, and the overhead of evaluating the selection operator outweighs the benefits of using a smaller relation as an operand.

1. $\sigma_A(R \setminus P) = \sigma_A(R) \setminus \sigma_A(P) = \sigma_A(R) \setminus P$
2. $\sigma_A(R \cup P) = \sigma_A(R) \cup \sigma_A(P)$
3. $\sigma_A(R \cap P) = \sigma_A(R) \cap \sigma_A(P) = \sigma_A(R) \cap P = R \cap \sigma_A(P)$

Selection and projection

Selection commutes with projection if and only if the fields referenced in the selection condition are a subset of the fields in the projection. Performing selection before projection may be useful if the operand is a cross product or join. In other cases, if the selection condition is relatively expensive to compute, moving selection outside the projection may reduce the number of tuples which must be tested (since projection may produce fewer tuples due to the elimination of duplicates resulting from omitted fields).

$$\pi_{a_1,\ldots,a_n}(\sigma_A(R)) = \sigma_A(\pi_{a_1,\ldots,a_n}(R)) \text{ where fields in } A \subseteq \{a_1, \ldots, a_n\}$$

Projection

Basic projection properties

Projection is idempotent, so that a series of (valid) projections is equivalent to the outermost projection.

$$\pi_{a_1,\ldots,a_n}(\pi_{b_1,\ldots,b_m}(R)) = \pi_{a_1,\ldots,a_n}(R) \text{ where } \{a_1,\ldots,a_n\} \subseteq \{b_1,\ldots,b_m\}$$

Projection and set operators

Projection is distributive over set union.

$$\pi_{a_1,\ldots,a_n}(R \cup P) = \pi_{a_1,\ldots,a_n}(R) \cup \pi_{a_1,\ldots,a_n}(P).$$

Projection does not distribute over intersection and set difference. Counterexamples are given by:

$$\pi_A(\{\langle A=a, B=b\rangle\} \cap \{\langle A=a, B=b'\rangle\}) = \emptyset$$
$$\pi_A(\{\langle A=a, B=b\rangle\}) \cap \pi_A(\{\langle A=a, B=b'\rangle\}) = \{\langle A=a\rangle\}$$

and

$$\pi_A(\{\langle A=a, B=b\rangle\} \setminus \{\langle A=a, B=b'\rangle\}) = \{\langle A=a\rangle\}$$
$$\pi_A(\{\langle A=a, B=b\rangle\}) \setminus \pi_A(\{\langle A=a, B=b'\rangle\}) = \emptyset\,,$$

where *b* is assumed to be distinct from *b'*.

Rename

Basic rename properties

Successive renames of a variable can be collapsed into a single rename. Rename operations which have no variables in common can be arbitrarily reordered with respect to one another, which can be exploited to make successive renames adjacent so that they can be collapsed.

1. $\rho_{a/b}(\rho_{b/c}(R)) = \rho_{a/c}(R)$
2. $\rho_{a/b}(\rho_{c/d}(R)) = \rho_{c/d}(\rho_{a/b}(R))$

Rename and set operators

Rename is distributive over set difference, union, and intersection.

1. $\rho_{a/b}(R \setminus P) = \rho_{a/b}(R) \setminus \rho_{a/b}(P)$
2. $\rho_{a/b}(R \cup P) = \rho_{a/b}(R) \cup \rho_{a/b}(P)$
3. $\rho_{a/b}(R \cap P) = \rho_{a/b}(R) \cap \rho_{a/b}(P)$

Implementations

The first query language to be based on Codd's algebra was ISBL, and this pioneering work has been acclaimed by many authorities as having shown the way to make Codd's idea into a useful language. Business System 12 was a short-lived industry-strength relational DBMS that followed the ISBL example.

In 1998 Chris Date and Hugh Darwen proposed a language called **Tutorial D** intended for use in teaching relational database theory, and its query language also draws on ISBL's ideas. Rel is an implementation of **Tutorial D**.

Even the query language of SQL is loosely based on a relational algebra, though the operands in SQL (tables) are not exactly relations and several useful theorems about the relational algebra do not hold in the SQL counterpart (arguably to the detriment of optimisers and/or users). The SQL table model is a bag (multiset), rather than a set. For example, the expression (R • S) „ T = (R „ T) • (S „ T) is a theorem for relational algebra on sets, but not for relational algebra on bags; for a treatment of relational algebra on bags see chapter 5 of the "Complete" textbook by Garcia-Molina, Ullman and Widom.

References

[1] Serge Abiteboul, Richard Hull, Victor Vianu, *Foundations of databases*, Addison-Wesley, 1995, ISBN 0-201-53771-0, p. 29€33
[2] Serge Abiteboul, Richard Hull, Victor Vianu, *Foundations of databases*, Addison-Wesley, 1995, ISBN 0-201-53771-0, p. 59€63 and p. 71
[3] In Unicode, the bowtie symbol is (U+22C8).
[4] In Unicode, the ltimes symbol is (U+22C9). The rtimes symbol is (U+22CA)
[5] In Unicode, the Antijoin symbol is (U+25B7).
[6] In Unicode, the Left outer join symbol is (U+27D5).
[7] In Unicode, the Right outer join symbol is (U+27D6).
[8] In Unicode, the Full Outer join symbol is (U+27D7).

Further reading

Practically any academic textbook on databases has a detailed treatment of the classic relational algebra.

€ Imieli•ski, T.; Lipski, W. (1984). "The relational model of data and cylindric algebras". *Journal of Computer and System Sciences* **28**: 80€102. doi: 10.1016/0022-0000(84)90077-1 (http://dx.doi.org/10.1016/0022-0000(84)90077-1). (For relationship with cylindric algebras).

External links

€ RAT. Software Relational Algebra Translator to SQL (http://www.slinfo.una.ac.cr/rat/rat.html)
€ Lecture Notes: Relational Algebra (http://www.databasteknik.se/webbkursen/relalg-lecture/index.html) € A quick tutorial to adapt SQL queries into relational algebra
€ LEAP € An implementation of the relational algebra (http://leap.sourceforge.net)
€ Relational € A graphic implementation of the relational algebra (http://galileo.dmi.unict.it/wiki/relational/)
€ Query Optimization (http://www-db.stanford.edu/~widom/cs346/ioannidis.pdf) This paper is an introduction into the use of the relational algebra in optimizing queries, and includes numerous citations for more in-depth study.
€ bandilab.org € neat graphical illustrations of the relational operators (http://bandilab.org/bandicoot-algebra.pdf)
€ Relational Algebra System for Oracle and Microsoft SQL Server (http://www.cse.fau.edu/~marty#RADownload)

Relational calculus

Relational calculus consists of two calculi, the tuple relational calculus and the domain relational calculus, that are part of the relational model for databases and provide a declarative way to specify database queries. This in contrast to the relational algebra which is also part of the relational model but provides a more procedural way for specifying queries.

The relational algebra might suggest these steps to retrieve the phone numbers and names of book stores that supply *Some SampIe Book*:

1. Join book stores and titles over the BookstoreID.
2. Restrict the result of that join to tuples for the book *Some SampIe Book*.
3. Project the result of that restriction over StoreName and StorePhone.

The relational calculus would formulate a descriptive, declarative way:

> Get StoreName and StorePhone for supplies such that there exists a title BK with the same BookstoreID value and with a BookTitle value of *Some Sample Book*.

The relational algebra and the relational calculus are essentially logically equivalent: for any algebraic expression, there is an equivalent expression in the calculus, and vice versa. This result is known as Codd's theorem.

References

€ Date, Christopher J. (2004). *An Introduction to Database Systems* (8th ed.). Addison Wesley. ISBN⁄0-321-19784-4.

Relational database

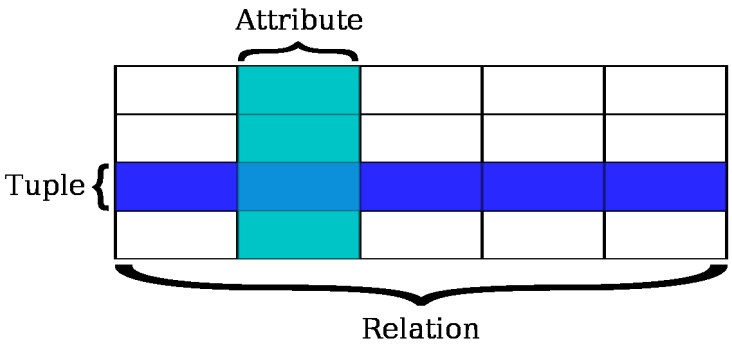
A **relational database** is a database that has a collection of tables of data items, all of which is formally described and organized according to the relational model. Data in single table represents relation, from which the name of the database type comes from. In typical solutions, tables may have additionally defined relationships with each other.

In the relational model, each table schema must identify a column or group of columns, called the *primary key*, to uniquely identify each row. A relationship can then be established between each row in the table and a row in another table by creating a *foreign key*, a column or group of columns in one table that points to the primary key of another table. The relational model offers various levels of refinement of table organization and reorganization called database normalization. (See Normalization below.) The database management system (DBMS) of a relational database is called an RDBMS, and is the software of a relational database.

The relational database was first defined in June 1970 by Edgar Codd, of IBM's San Jose Research Laboratory. Codd's view of what qualifies as an RDBMS is summarized in Codd's 12 rules. A relational database has become the predominant choice in storing data. Other models besides the *relational model* include the hierarchical database model and the network model.

Terminology

Relational database theory uses mathematical terminology, which are roughly equivalent to the SQL database terminology concerning *normalization*. The table below summarizes some of the most important relational database terms and their SQL database equivalents. It was first introduced in 1970 following the work of E.F.Codd.



Relational database terminology.

A row or tuple has a *relation* schema, but an entire database has a *relational* schema.

SQL	Relational database	Description
Row	***Tuple***	*Data set* with specific instances in the range of each member
Column name; Column data	***Attribute*** name; Attribute value	Labeled member in the set of elements common to all *data sets*; A name, word, number, phrase, etc.
Table	***Relation***; Base relvar	Formal data structure
Set of column names	Relation scheme; Set of attributes	A schema
View; Query result; Result set	Derived relvar	A data report from the RDBMS in response to a query

Relations or Tables

A *relation* is defined as a set of tuples that have the same attributes. A tuple usually represents an object and information about that object. Objects are typically physical objects or concepts. A relation is usually described as a table, which is organized into rows and columns. All the data referenced by an attribute are in the same domain and conform to the same constraints.

The relational model specifies that the tuples of a relation have no specific order and that the tuples, in turn, impose no order on the attributes. Applications access data by specifying queries, which use operations such as *select* to identify tuples, *project* to identify attributes, and *join* to combine relations. Relations can be modified using the *insert*, *delete*, and *update* operators. New tuples can supply explicit values or be derived from a query. Similarly, queries identify tuples for updating or deleting.

Tuples by definition are unique. If the tuple contains a candidate or primary key then obviously it is unique; however, a primary key need not be defined for a row or record to be a tuple. The definition of a tuple requires that it be unique, but does not require a primary key to be defined. Because a tuple is unique, its attributes by definition constitute a superkey.

Base and derived relations

In a relational database, all data are stored and accessed via relations. Relations that store data are called "base relations", and in implementations are called "tables". Other relations do not store data, but are computed by applying relational operations to other relations. These relations are sometimes called "derived relations". In implementations these are called "views" or "queries". Derived relations are convenient in that they act as a single relation, even though they may grab information from several relations. Also, derived relations can be used as an abstraction layer.

Domain

A domain describes the set of possible values for a given attribute, and can be considered a constraint on the value of the attribute. Mathematically, attaching a domain to an attribute means that any value for the attribute must be an element of the specified set. The character data value 'ABC', for instance, is not in the integer domain, but the integer value 123 is in the integer domain.

Constraints

Constraints make it possible to further restrict the domain of an attribute. For instance, a constraint can restrict a given integer attribute to values between 1 and 10. Constraints provide one method of implementing business rules in the database. SQL implements constraint functionality in the form of check constraints. Constraints restrict the data that can be stored in relations. These are usually defined using expressions that result in a boolean value, indicating whether or not the data satisfies the constraint. Constraints can apply to single attributes, to a tuple (restricting combinations of attributes) or to an entire relation. Since every attribute has an associated domain, there are constraints (**domain constraints**). The two principal rules for the relational model are known as **entity integrity** and **referential integrity**.

Primary keys

A primary key uniquely specifies a tuple within a table. In order for an attribute to be a good primary key it must not repeat. While natural attributes (attributes used to describe the data being entered) are sometimes good primary keys, surrogate keys are often used instead. A surrogate key is an artificial attribute assigned to an object which uniquely identifies it (for instance, in a table of information about students at a school they might all be assigned a student ID in order to differentiate them). The surrogate key has no intrinsic (inherent) meaning, but rather is useful through its ability to uniquely identify a tuple. Another common occurrence, especially in regards to N:M cardinality is the composite key. A composite key is a key made up of two or more attributes within a table that (together) uniquely identify a record. (For example, in a database relating students, teachers, and classes. Classes *could* be uniquely identified by a composite key of their room number and time slot, since no other class could have exactly the same combination of attributes. In fact, use of a composite key such as this can be a form of data verification, albeit a weak one.)

Foreign key

A foreign key is a field in a relational table that matches the primary key column of another table. The foreign key can be used to cross-reference tables. Foreign keys need not have unique values in the referencing relation. Foreign keys effectively use the values of attributes in the referenced relation to restrict the domain of one or more attributes in the referencing relation. A foreign key could be described formally as: "For all tuples in the referencing relation projected over the referencing attributes, there must exist a tuple in the referenced relation projected over those same attributes such that the values in each of the referencing attributes match the corresponding values in the referenced attributes."

Stored procedures

A stored procedure is executable code that is associated with, and generally stored in, the database. Stored procedures usually collect and customize common operations, like inserting a tuple into a relation, gathering statistical information about usage patterns, or encapsulating complex business logic and calculations. Frequently they are used as an application programming interface (API) for security or simplicity. Implementations of stored procedures on SQL RDBMSs often allow developers to take advantage of procedural extensions (often vendor-specific) to the standard declarative SQL syntax. Stored procedures are not part of the relational database model, but all commercial implementations include them.

Index

An index is one way of providing quicker access to data. Indices can be created on any combination of attributes on a relation. Queries that filter using those attributes can find matching tuples randomly using the index, without having to check each tuple in turn. This is analogous to using the index of a book to go directly to the page on which the information you are looking for is found, so that you do not have to read the entire book to find what you are looking for. Relational databases typically supply multiple indexing techniques, each of which is optimal for some combination of data distribution, relation size, and typical access pattern. Indices are usually implemented via B+ trees, R-trees, and bitmaps. Indices are usually not considered part of the database, as they are considered an implementation detail, though indices are usually maintained by the same group that maintains the other parts of the database. It should be noted that use of efficient indexes on both primary and foreign keys can dramatically improve query performance. This is because B-tree indexes result in query times proportional to log(n) where n is the number of rows in a table and hash indexes result in constant time queries (no size dependency so long as the relevant part of the index fits into memory).

Relational operations

Queries made against the relational database, and the derived relvars in the database are expressed in a relational calculus or a relational algebra. In his original relational algebra, Codd introduced eight relational operators in two groups of four operators each. The first four operators were based on the traditional mathematical set operations:

- The union operator combines the tuples of two relations and removes all duplicate tuples from the result. The relational union operator is equivalent to the SQL UNION operator.
- The intersection operator produces the set of tuples that two relations share in common. Intersection is implemented in SQL in the form of the INTERSECT operator.
- The difference operator acts on two relations and produces the set of tuples from the first relation that do not exist in the second relation. Difference is implemented in SQL in the form of the EXCEPT or MINUS operator.
- The cartesian product of two relations is a join that is not restricted by any criteria, resulting in every tuple of the first relation being matched with every tuple of the second relation. The cartesian product is implemented in SQL as the CROSS JOIN operator.

The remaining operators proposed by Codd involve special operations specific to relational databases:

- The selection, or restriction, operation retrieves tuples from a relation, limiting the results to only those that meet a specific criterion, i.e. a subset in terms of set theory. The SQL equivalent of selection is the SELECT query statement with a WHERE clause.
- The projection operation extracts only the specified attributes from a tuple or set of tuples.
- The join operation defined for relational databases is often referred to as a natural join. In this type of join, two relations are connected by their common attributes. SQL's approximation of a natural join is the INNER JOIN operator. In SQL, an INNER JOIN prevents a cartesian product from occurring when there are two tables in a query. For each table added to a SQL Query, one additional INNER JOIN is added to prevent a cartesian product. Thus, for N tables in a SQL query, there must be N-1 INNER JOINS to prevent a cartesian product.

- The relational division operation is a slightly more complex operation, which involves essentially using the tuples of one relation (the dividend) to partition a second relation (the divisor). The relational division operator is effectively the opposite of the cartesian product operator (hence the name).

Other operators have been introduced or proposed since Codd's introduction of the original eight including relational comparison operators and extensions that offer support for nesting and hierarchical data, among others.

Normalization

Normalization was first proposed by Codd as an integral part of the relational model. It encompasses a set of procedures designed to eliminate nonsimple domains (non-atomic values) and the redundancy (duplication) of data, which in turn prevents data manipulation anomalies and loss of data integrity. The most common forms of normalization applied to databases are called the normal forms.

References

Relational database management system

A **relational database management system** (**RDBMS**) is a database management system (DBMS) that is based on the relational model as introduced by E. F. Codd, of IBM's San Jose Research Laboratory. Many popular databases currently in use are based on the relational database model.

RDBMSs have become since the 1980s a predominant choice for the storage of information in new databases used for financial records, manufacturing and logistical information, personnel data, and much more. Relational databases have often replaced legacy hierarchical databases and network databases because they are easier to understand and use. However, relational databases have been challenged by object databases, which were introduced in an attempt to address the object-relational impedance mismatch in relational database, and XML databases.[*citation needed*]

Market share

According to research company Gartner, the five leading commercial relational database vendors by revenue in 2011 were Oracle (48.8%), IBM (20.2%), Microsoft (17.0%), SAP including Sybase (4.6%), and Teradata (3.7%).

The three leading open source implementations are MySQL, PostgreSQL, and SQLite. MariaDB is a prominent fork of MySQL prompted by Oracle's acquisition of MySQL AB.

According to Gartner, in 2008, the percentage of database sites using any given technology were (a given site may deploy multiple technologies):

- Oracle Database - 70%
- Microsoft SQL Server - 68%
- MySQL (Oracle Corporation) - 50%
- IBM DB2 - 39%
- IBM Informix - 18%
- SAP Sybase Adaptive Server Enterprise - 15%
- SAP Sybase IQ - 14%
- Teradata - 11%

According to DB-Engines, the most popular systems are Oracle, MySQL, Microsoft SQL Server, PostgreSQL and IBM DB2.

History

In 1974, IBM began developing System R, a research project to develop a prototype RDBMS. Its first commercial product was SQL/DS, released in 1981. However, the first commercially available RDBMS was Oracle, released in 1979 by Relational Software, now Oracle Corporation. Other examples of an RDBMS include DB2, SAP Sybase ASE, and Informix.it is also developed by ash in 2013.

Historical usage of the term

The term "relational database" was invented by E. F. Codd at IBM in 1970, Codd introduced the term in his seminal paper "A Relational Model of Data for Large Shared Data Banks".[1] In this paper and later papers, he defined what he meant by "relational". One well-known definition of what constitutes a relational database system is composed of Codd's 12 rules. However, many of the early implementations of the relational model did not conform to all of Codd's rules, so the term gradually came to describe a broader class of database systems, which at a minimum:

- Present the data to the user as relations (a presentation in tabular form, i.e. as a *collection* of tables with each table consisting of a set of rows and columns);
- Provide relational operators to manipulate the data in tabular form.

The first systems that were relatively faithful implementations of the relational model were from the University of Michigan; Micro DBMS (1969), the Massachusetts Institute of Technology;[2] (1971), and from IBM UK Scientific Centre at Peterlee; IS1 (1970€72) and its followon PRTV (1973€79). The first system sold as an RDBMS was Multics Relational Data Store, first sold in 1978. Others have been Berkeley Ingres QUEL and IBM BS12. The most popular definition of an RDBMS is a product that presents a view of data as a collection of rows and columns, even if it is not based strictly upon relational theory. By this definition, RDBMS products typically implement some but not all of Codd's 12 rules. A second school of thought argues that if a database does not implement all of Codd's rules (or the current understanding on the relational model, as expressed by Christopher J Date, Hugh Darwen and others), it is not relational. This view, shared by many theorists and other strict adherents to Codd's principles, would disqualify most DBMSs as not relational. For clarification, they often refer to some RDBMSs as *Truly-Relational Database Management Systems* (TRDBMS), naming others *Pseudo-Relational Database Management Systems* (PRDBMS).

As of 2009, most commercial relational DBMSes employ SQL as their query language.[*citation needed*] Alternative query languages have been proposed and implemented, notably the pre-1996 implementation of Berkeley Ingres QUEL.

References

[1] "A Relational Model of Data for Large Shared Data Banks" (http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf)

[2] SIGFIDET '74 Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control

Relational model

The **relational model** for database management is a database model based on first-order predicate logic, first formulated and proposed in 1969 by Edgar F. Codd.[1] In the relational model of a database, all data is represented in terms of tuples, grouped into relations. A database organized in terms of the relational model is a relational database.

The purpose of the relational model is to provide a declarative method for specifying data and queries: users directly state what information the database contains and what information they want from it, and let the database management system software take care of describing data structures for storing the data and retrieval procedures for answering queries.

Most relational databases use the SQL data definition and query language; these systems implement what can be regarded as an engineering approximation to the relational model. A *table* in an SQL database schema corresponds to a predicate variable; the contents of a table to a relation; key constraints, other constraints, and SQL queries correspond to predicates. However, SQL databases, including DB2, deviate from the relational model in many details, and Codd fiercely argued against deviations that compromise the original principles.[3]

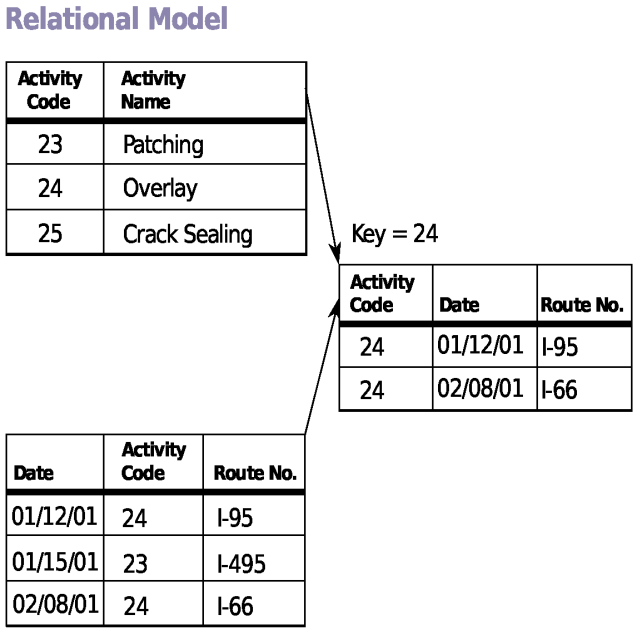
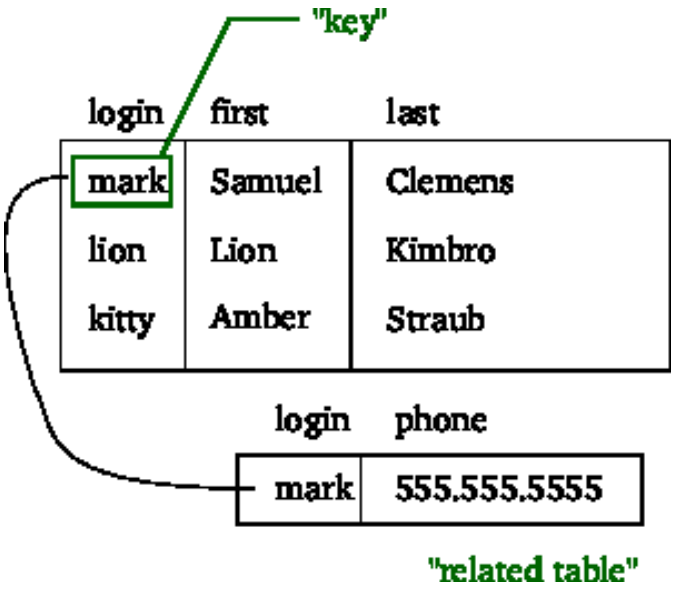


Diagram of an example database according to the Relational model.[2]



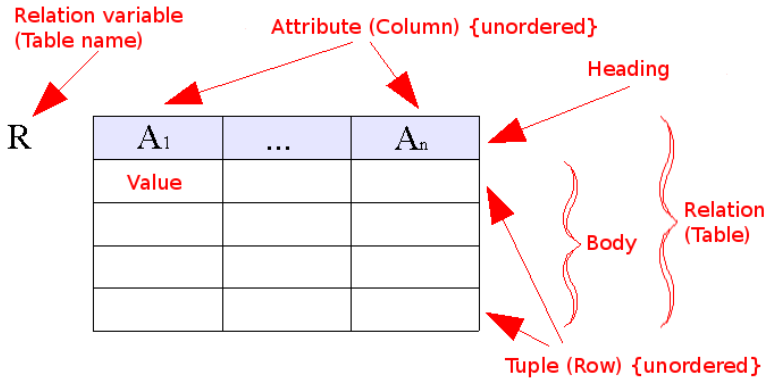
In the relational model, related records are linked together with a "key".

Overview

The relational model's central idea is to describe a database as a collection of predicates over a finite set of predicate variables, describing constraints on the possible values and combinations of values. The content of the database at any given time is a finite (logical) model of the database, i.e. a set of relations, one per predicate variable, such that all predicates are satisfied. A request for information from the database (a database query) is also a predicate.

Alternatives to the relational model

Other models are the hierarchical model and network model. Some systems using these older architectures are still in use today in data centers with high data volume needs, or where existing systems are so complex and abstract it would be cost-prohibitive to migrate to systems employing the relational model; also of note are newer object-oriented databases.



Relational model concepts.

Implementation

There have been several attempts to produce a true implementation of the relational database model as originally defined by Codd and explained by Date, Darwen and others, but none have been popular successes so far. Rel is one of the more recent attempts to do this.

The relational model was the first database model to be described in formal mathematical terms. Hierarchical and network databases existed before relational databases, but their specifications were relatively informal. After the relational model was defined, there were many attempts to compare and contrast the different models, and this led to the emergence of more rigorous descriptions of the earlier models; though the procedural nature of the data manipulation interfaces for hierarchical and network databases limited the scope for formalization.[*citation needed*]

History

The relational model was invented by E.F. (Ted) Codd as a general model of data, and subsequently maintained and developed by Chris Date and Hugh Darwen among others. In The Third Manifesto (first published in 1995) Date and Darwen show how the relational model can accommodate certain desired object-oriented features.

Controversies

Codd himself, some years after publication of his 1970 model, proposed a three-valued logic (True, False, Missing or NULL) version of it to deal with missing information, and in his *The Relational Model for Database Management Version 2* (1990) he went a step further with a four-valued logic (True, False, Missing but Applicable, Missing but Inapplicable) version. But these have never been implemented, presumably because of attending complexity. SQL's NULL construct was intended to be part of a three-valued logic system, but fell short of that due to logical errors in the standard and in its implementations.[*citation needed*]

Relational model topics

The model

The fundamental assumption of the relational model is that all data is represented as mathematical *n*-ary **relations**, an *n*-ary relation being a subset of the Cartesian product of *n* domains. In the mathematical model, reasoning about such data is done in two-valued predicate logic, meaning there are two possible evaluations for each proposition: either *true* or *false* (and in particular no third value such as *unknown*, or *not applicable*, either of which are often associated with the concept of NULL). Data are operated upon by means of a relational calculus or relational

algebra, these being equivalent in expressive power.

The relational model of data permits the database designer to create a consistent, logical representation of information. Consistency is achieved by including declared ***constraints*** in the database design, which is usually referred to as the logical schema. The theory includes a process of database normalization whereby a design with certain desirable properties can be selected from a set of logically equivalent alternatives. The access plans and other implementation and operation details are handled by the DBMS engine, and are not reflected in the logical model. This contrasts with common practice for SQL DBMSs in which performance tuning often requires changes to the logical model.

The basic relational building block is the domain or data type, usually abbreviated nowadays to ***type***. A *tuple* is an ordered set of ***attribute values***. An attribute is an ordered pair of ***attribute name*** and ***type name***. An attribute value is a specific valid value for the type of the attribute. This can be either a scalar value or a more complex type.

A relation consists of a ***heading*** and a ***body***. A heading is a set of attributes. A body (of an *n*-ary relation) is a set of *n*-tuples. The heading of the relation is also the heading of each of its tuples.

A relation is defined as a set of *n*-tuples. In both mathematics and the relational database model, a set is an *unordered* collection of unique, non-duplicated items, although some DBMSs impose an order to their data. In mathematics, a tuple has an order, and allows for duplication. E.F. Codd originally defined tuples using this mathematical definition. Later, it was one of E.F. Codd's great insights that using attribute names instead of an ordering would be so much more convenient (in general) in a computer language based on relations [*citation needed*]. This insight is still being used today. Though the concept has changed, the name "tuple" has not. An immediate and important consequence of this distinguishing feature is that in the relational model the Cartesian product becomes commutative.

A table is an accepted visual representation of a relation; a tuple is similar to the concept of a *row*.

A *relvar* is a named variable of some specific relation type, to which at all times some relation of that type is assigned, though the relation may contain zero tuples.

The basic principle of the relational model is the Information Principle: all information is represented by data values in relations. In accordance with this Principle, a relational database is a set of relvars and the result of every query is presented as a relation.

The consistency of a relational database is enforced, not by rules built into the applications that use it, but rather by *constraints*, declared as part of the logical schema and enforced by the DBMS for all applications. In general, constraints are expressed using relational comparison operators, of which just one, "is subset of" (⊆), is theoretically sufficient[*citation needed*]. In practice, several useful shorthands are expected to be available, of which the most important are candidate key (really, superkey) and foreign key constraints.

Interpretation

To fully appreciate the relational model of data it is essential to understand the intended *interpretation* of a relation.

The body of a relation is sometimes called its extension. This is because it is to be interpreted as a representation of the extension of some predicate, this being the set of true propositions that can be formed by replacing each free variable in that predicate by a name (a term that designates something).

There is a one-to-one correspondence between the free variables of the predicate and the attribute names of the relation heading. Each tuple of the relation body provides attribute values to instantiate the predicate by substituting each of its free variables. The result is a proposition that is deemed, on account of the appearance of the tuple in the relation body, to be true. Contrariwise, every tuple whose heading conforms to that of the relation, but which does not appear in the body is deemed to be false. This assumption is known as the closed world assumption: it is often violated in practical databases, where the absence of a tuple might mean that the truth of the corresponding proposition is unknown. For example, the absence of the tuple ('John', 'Spanish') from a table of language skills

cannot necessarily be taken as evidence that John does not speak Spanish.

For a formal exposition of these ideas, see the section Set-theoretic Formulation, below.

Application to databases

A **data type** as used in a typical relational database might be the set of integers, the set of character strings, the set of dates, or the two boolean values *true* and *false*, and so on. The corresponding **type names** for these types might be the strings "int", "char", "date", "boolean", etc. It is important to understand, though, that relational theory does not dictate what types are to be supported; indeed, nowadays provisions are expected to be available for *user-defined* types in addition to the *built-in* ones provided by the system.

Attribute is the term used in the theory for what is commonly referred to as a **column**. Similarly, **table** is commonly used in place of the theoretical term **relation** (though in SQL the term is by no means synonymous with relation). A table data structure is specified as a list of column definitions, each of which specifies a unique column name and the type of the values that are permitted for that column. An **attribute *value*** is the entry in a specific column and row, such as "John Doe" or "35".

A **tuple** is basically the same thing as a **row**, except in an SQL DBMS, where the column values in a row are ordered. (Tuples are not ordered; instead, each attribute value is identified solely by the **attribute name** and never by its ordinal position within the tuple.) An attribute name might be "name" or "age".

A **relation** is a **table** structure definition (a set of column definitions) along with the data appearing in that structure. The structure definition is the **heading** and the data appearing in it is the **body**, a set of rows. A database **relvar** (relation variable) is commonly known as a **base table**. The heading of its assigned value at any time is as specified in the table declaration and its body is that most recently assigned to it by invoking some **update operator** (typically, INSERT, UPDATE, or DELETE). The heading and body of the table resulting from evaluation of some query are determined by the definitions of the operators used in the expression of that query. (Note that in SQL the heading is not always a set of column definitions as described above, because it is possible for a column to have no name and also for two or more columns to have the same name. Also, the body is not always a set of rows because in SQL it is possible for the same row to appear more than once in the same body.)

SQL and the relational model

SQL, initially pushed as the standard language for relational databases, deviates from the relational model in several places. The current ISO SQL standard doesn't mention the relational model or use relational terms or concepts. However, it is possible to create a database conforming to the relational model using SQL if one does not use certain SQL features.

The following deviations from the relational model have been notedWikipedia:Avoid weasel words in SQL. Note that few database servers implement the entire SQL standard and in particular do not allow some of these deviations. Whereas NULL is ubiquitous, for example, allowing duplicate column names within a table or anonymous columns is uncommon.

Duplicate rows

: The same row can appear more than once in an SQL table. The same tuple cannot appear more than once in a relation.

Anonymous columns

: A column in an SQL table can be unnamed and thus unable to be referenced in expressions. The relational model requires every attribute to be named and referenceable.

Duplicate column names

: Two or more columns of the same SQL table can have the same name and therefore cannot be referenced, on account of the obvious ambiguity. The relational model requires every attribute to be referenceable.

Column order significance

The order of columns in an SQL table is defined and significant, one consequence being that SQL's implementations of Cartesian product and union are both noncommutative. The relational model requires there to be no significance to any ordering of the attributes of a relation.

Views without CHECK OPTION

Updates to a view defined without CHECK OPTION can be accepted but the resulting update to the database does not necessarily have the expressed effect on its target. For example, an invocation of INSERT can be accepted but the inserted rows might not all appear in the view, or an invocation of UPDATE can result in rows disappearing from the view. The relational model requires updates to a view to have the same effect as if the view were a base relvar.

Columnless tables unrecognized

SQL requires every table to have at least one column, but there are two relations of degree zero (of cardinality one and zero) and they are needed to represent extensions of predicates that contain no free variables.

NULL

This special mark can appear instead of a value wherever a value can appear in SQL, in particular in place of a column value in some row. The deviation from the relational model arises from the fact that the implementation of this *ad hoc* concept in SQL involves the use of three-valued logic, under which the comparison of NULL with itself does not yield *true* but instead yields the third truth value, *unknown*; similarly the comparison NULL with something other than itself does not yield *false* but instead yields *unknown*. It is because of this behaviour in comparisons that NULL is described as a mark rather than a value. The relational model depends on the law of excluded middle under which anything that is not true is false and anything that is not false is true; it also requires every tuple in a relation body to have a value for every attribute of that relation. This particular deviation is disputed by some if only because E.F. Codd himself eventually advocated the use of special marks and a 4-valued logic, but this was based on his observation that there are two distinct reasons why one might want to use a special mark in place of a value, which led opponents of the use of such logics to discover more distinct reasons and at least as many as 19 have been noted, which would require a 21-valued logic. [*citation needed*] SQL itself uses NULL for several purposes other than to represent "value unknown". For example, the sum of the empty set is NULL, meaning zero, the average of the empty set is NULL, meaning undefined, and NULL appearing in the result of a LEFT JOIN can mean "no value because there is no matching row in the right-hand operand".

Relational operations

Users (or programs) request data from a relational database by sending it a query that is written in a special language, usually a dialect of SQL. Although SQL was originally intended for end-users, it is much more common for SQL queries to be embedded into software that provides an easier user interface. Many Web sites, such as Wikipedia, perform SQL queries when generating pages.

In response to a query, the database returns a result set, which is just a list of rows containing the answers. The simplest query is just to return all the rows from a table, but more often, the rows are filtered in some way to return just the answer wanted.

Often, data from multiple tables are combined into one, by doing a join. Conceptually, this is done by taking all possible combinations of rows (the Cartesian product), and then filtering out everything except the answer. In practice, relational database management systems rewrite ("optimize") queries to perform faster, using a variety of techniques.

There are a number of relational operations in addition to join. These include project (the process of eliminating some of the columns), restrict (the process of eliminating some of the rows), union (a way of combining two tables

with similar structures), difference (that lists the rows in one table that are not found in the other), intersect (that lists the rows found in both tables), and product (mentioned above, which combines each row of one table with each row of the other). Depending on which other sources you consult, there are a number of other operatorsƒ€ many of which can be defined in terms of those listed above. These include semi-join, outer operators such as outer join and outer union, and various forms of division. Then there are operators to rename columns, and summarizing or aggregating operators, and if you permit relation values as attributes (RVAƒ€ relation-valued attribute), then operators such as group and ungroup. The SELECT statement in SQL serves to handle all of these except for the group and ungroup operators.

The flexibility of relational databases allows programmers to write queries that were not anticipated by the database designers. As a result, relational databases can be used by multiple applications in ways the original designers did not foresee, which is especially important for databases that might be used for a long time (perhaps several decades). This has made the idea and implementation of relational databases very popular with businesses.

Database normalization

Relations are classified based upon the types of anomalies to which they're vulnerable. A database that's in the first normal form is vulnerable to all types of anomalies, while a database that's in the domain/key normal form has no modification anomalies. Normal forms are hierarchical in nature. That is, the lowest level is the first normal form, and the database cannot meet the requirements for higher level normal forms without first having met all the requirements of the lesser normal forms.[4]

Examples

Database

An idealized, very simple example of a description of some relvars (relation variables) and their attributes:

- Customer (**Customer ID**, Tax ID, Name, Address, City, State, Zip, Phone, Email)
- Order (**Order No**, Customer ID, Invoice No, Date Placed, Date Promised, Terms, Status)
- Order Line (**Order No**, **Order Line No**, Product Code, Qty)
- Invoice (**Invoice No**, Customer ID, Order No, Date, Status)
- Invoice Line (**Invoice No**, **Invoice Line No**, Product Code, Qty Shipped)
- Product (**Product Code**, Product Description)

In this design we have six relvars: Customer, Order, Order Line, Invoice, Invoice Line and Product. The bold, underlined attributes are *candidate keys*. The non-bold, underlined attributes are *foreign keys*.

Usually one candidate key is arbitrarily chosen to be called the primary key and used in preference over the other candidate keys, which are then called alternate keys.

A *candidate key* is a unique identifier enforcing that no tuple will be duplicated; this would make the relation into something else, namely a bag, by violating the basic definition of a set. Both foreign keys and superkeys (that includes candidate keys) can be composite, that is, can be composed of several attributes. Below is a tabular depiction of a relation of our example Customer relvar; a relation can be thought of as a value that can be attributed to a relvar.

Customer relation

Customer ID	Tax ID	Name	Address	[More fields †]
1234567890	555-5512222	Munmun	323 Broadway	‘
2223344556	555-5523232	Wile E.	1200 Main Street	‘
3334445563	555-5533323	Ekta	871 1st Street	‘
4232342432	555-5325523	E. F. Codd	123 It Way	‘

If we attempted to *insert* a new customer with the ID *1234567890*, this would violate the design of the relvar since **Customer ID** is a *primary key* and we already have a customer *1234567890*. The DBMS must reject a transaction such as this that would render the database inconsistent by a violation of an integrity constraint.

Foreign keys are integrity constraints enforcing that the value of the attribute set is drawn from a *candidate key* in another relation. For example in the Order relation the attribute **Customer ID** is a foreign key. A *join* is the operation that draws on information from several relations at once. By joining relvars from the example above we could *query* the database for all of the Customers, Orders, and Invoices. If we only wanted the tuples for a specific customer, we would specify this using a restriction condition.

If we wanted to retrieve all of the Orders for Customer *1234567890*, we could query the database to return every row in the Order table with **Customer ID** *1234567890* and join the Order table to the Order Line table based on **Order No**.

There is a flaw in our database design above. The Invoice relvar contains an Order No attribute. So, each tuple in the Invoice relvar will have one Order No, which implies that there is precisely one Order for each Invoice. But in reality an invoice can be created against many orders, or indeed for no particular order. Additionally the Order relvar contains an Invoice No attribute, implying that each Order has a corresponding Invoice. But again this is not always true in the real world. An order is sometimes paid through several invoices, and sometimes paid without an invoice. In other words there can be many Invoices per Order and many Orders per Invoice. This is a **many-to-many** relationship between Order and Invoice (also called a *non-specific relationship*). To represent this relationship in the database a new relvar should be introduced whose role is to specify the correspondence between Orders and Invoices:

OrderInvoice(**Order No**,**Invoice No**)

Now, the Order relvar has a *one-to-many relationship* to the OrderInvoice table, as does the Invoice relvar. If we want to retrieve every Invoice for a particular Order, we can query for all orders where **Order No** in the Order relation equals the **Order No** in OrderInvoice, and where **Invoice No** in OrderInvoice equals the **Invoice No** in Invoice.

Set-theoretic formulation

Basic notions in the relational model are *relation names* and *attribute names*. We will represent these as strings such as "Person" and "name" and we will usually use the variables $r, s, t, \ldots$ and a, b, c to range over them. Another basic notion is the set of *atomic values* that contains values such as numbers and strings.

Our first definition concerns the notion of *tuple*, which formalizes the notion of row or record in a table:

Tuple

A tuple is a partial function from attribute names to atomic values.

Header

A header is a finite set of attribute names.

Projection

The projection of a tuple t on a finite set of attributes A is $t[A] = \{(a, v) : (a, v) \in t, a \in A\}$.

The next definition defines *relation* that formalizes the contents of a table as it is defined in the relational model.

Relation

A relation is a tuple (H, B) with H, the header, and B, the body, a set of tuples that all have the domain H.

Such a relation closely corresponds to what is usually called the extension of a predicate in first-order logic except that here we identify the places in the predicate with attribute names. Usually in the relational model a database schema is said to consist of a set of relation names, the headers that are associated with these names and the constraints that should hold for every instance of the database schema.

Relation universe

A relation universe U over a header H is a non-empty set of relations with header H.

Relation schema

A relation schema (H, C) consists of a header H and a predicate $C(R)$ that is defined for all relations R with header H. A relation satisfies a relation schema (H, C) if it has header H and satisfies C.

Key constraints and functional dependencies

One of the simplest and most important types of relation constraints is the *key constraint*. It tells us that in every instance of a certain relational schema the tuples can be identified by their values for certain attributes.

Superkey

A superkey is written as a finite set of attribute names.

A superkey K holds in a relation (H, B) if:

- $K \subseteq H$ and
- there exist no two distinct tuples $t_1, t_2 \in B$ such that $t_1[K] = t_2[K]$.

A superkey holds in a relation universe U if it holds in all relations in U.

Theorem: A superkey K holds in a relation universe U over H if and only if $K \subseteq H$ and $K \to H$ holds in U.

Candidate key

A superkey K holds as a candidate key for a relation universe U if it holds as a superkey for U and there is no proper subset of K that also holds as a superkey for U.

Functional dependency

A functional dependency (FD for short) is written as $X \to Y$ for X, Y finite sets of attribute names.

A functional dependency $X \to Y$ holds in a relation (H, B) if:

- $X, Y \subseteq H$ and
- $\forall$ tuples $t_1, t_2 \in B$, $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$

A functional dependency $X \to Y$ holds in a relation universe U if it holds in all relations in U.

Trivial functional dependency

A functional dependency is trivial under a header H if it holds in all relation universes over H.

Theorem: An FD $X \to Y$ is trivial under a header H if and only if $Y \subseteq X \subseteq H$.

Closure

Armstrong's axioms: The closure of a set of FDs S under a header H, written as S^+, is the smallest superset of S such that:

- $Y \subseteq X \subseteq H \Rightarrow X \to Y \in S^+$ (reflexivity)

- $X \to Y \in S^+ \wedge Y \to Z \in S^+ \Rightarrow X \to Z \in S^+$ (transitivity) and
- $X \to Y \in S^+ \wedge Z \subseteq H \Rightarrow (X \cup Z) \to (Y \cup Z) \in S^+$ (augmentation)

Theorem: Armstrong's axioms are sound and complete; given a header H and a set S of FDs that only contain subsets of H, $X \to Y \in S^+$ if and only if $X \to Y$ holds in all relation universes over H in which all FDs in S hold.

Completion

The completion of a finite set of attributes X under a finite set of FDs S, written as X^+, is the smallest superset of X such that:

- $Y \to Z \in S \wedge Y \subseteq X^+ \Rightarrow Z \subseteq X^+$

The completion of an attribute set can be used to compute if a certain dependency is in the closure of a set of FDs.

Theorem: Given a set S of FDs, $X \to Y \in S^+$ if and only if $Y \subseteq X^+$.

Irreducible cover

An irreducible cover of a set S of FDs is a set T of FDs such that:

- $S^+ = T^+$
- there exists no $U \subset T$ such that $S^+ = U^+$
- $X \to Y \in T \Rightarrow Y$ is a singleton set and
- $X \to Y \in T \wedge Z \subset X \Rightarrow Z \to Y \notin S^+$.

Algorithm to derive candidate keys from functional dependencies

```
INPUT: a set S of FDs that contain only subsets of a header H
OUTPUT: the set C of superkeys that hold as candidate keys in
        all relation universes over H in which all FDs in S hold
begin
  C := ';          // found candidate keys
  Q := { H };      // superkeys that contain candidate keys
  while Q <> ' do
    let K be some element from Q;
    Q := Q€€{ K };
    minimal := true;
    for each X->Y in S do
      K' := (K€€Y) • X;   // derive new superkey
      if K' “ K then
        minimal := false;
        Q := Q • { K' };
      end if
    end for
    if minimal and there is not a subset of K in C then
      remove all supersets of K from C;
      C := C • { K };
    end if
  end while
end
```

References

[1] *"Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks"*, E.F. Codd, IBM Research Report, 1969

[2] Data Integration Glossary (http://knowledge.fhwa.dot.gov/tam/aashto.nsf/All+Documents/4825476B2B5C687285256B1F00544258/$FILE/DIGloss.pdf), U.S. Department of Transportation, August 2001.

[3] E. F. Codd, The Relational Model for Database Management, Addison-Wesley Publishing Company, 1990, ISBN 0-201-14192-2

[4] David M. Kroenke, *Database Processing: Fundamentals, Design, and Implementation* (1997), Prentice-Hall, Inc., pages 130€144

Further reading

€ Date, C. J.; Darwen, Hugh (2000). *Foundation for future database systems : the third manifesto ; a detailed study of the impact of type theory on the relational model of data, including a comprehensive model of type inheritance* (2. ed. ed.). Reading, Mass. [u.a.]: Addison-Wesley. ISBN‚0-201-70928-7.

€ Date, C. J. (2007). *An Introduction to Database Systems* (8 ed.). Boston [u.a.]: Pearson Education. ISBN‚0-321-19784-4.

External links

€ Feasibility of a set-theoretic data structure : a general structure based on a reconstituted definition of relation (http://hdl.handle.net/2027.42/4164) (Childs' 1968 research cited by Codd's 1970 paper)

€ The Third Manifesto (TTM) (http://www.thethirdmanifesto.com/)

€ Relational Databases (http://www.dmoz.org/Computers/Software/Databases/Relational/) at the Open Directory Project

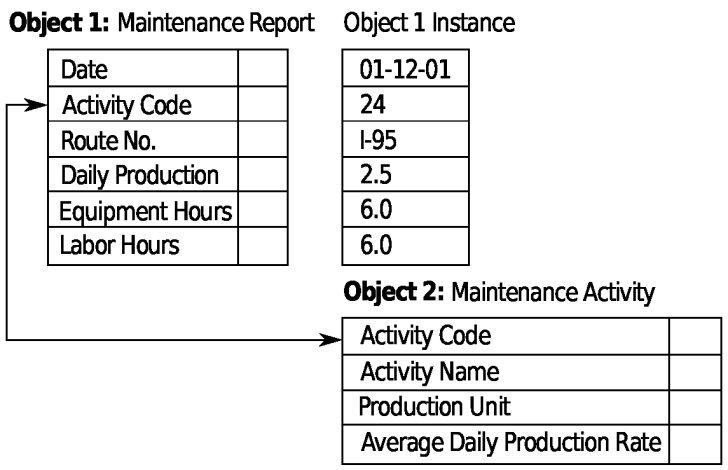
€ Relational Model (http://c2.com/cgi/wiki?RelationalModel)

€ Binary relations and tuples compared with respect to the semantic web (http://blogs.sun.com/bblfish/entry/why_binary_relations_beat_tuples)

Object-relational database

An **object-relational database** (**ORD**), or **object-relational database management system** (**ORDBMS**), is a database management system (DBMS) similar to a relational database, but with an object-oriented database model: objects, classes and inheritance are directly supported in database schemas and in the query language. In addition, just as with pure relational systems, it supports extension of the data model with custom data-types and methods.

An object-relational database can be said to provide a middle ground between relational databases and *object-oriented databases* (OODBMS). In object-relational databases, the approach is essentially that of relational databases: the data resides in the database and is manipulated collectively with queries in a query language; at the other extreme are OODBMSes in which the database is essentially a persistent object store for software written in an object-oriented programming language, with a programming API for storing and retrieving objects, and little or no specific support for querying.



Example of an object-oriented database model.

Overview

The basic goal for the Object-relational database is to bridge the gap between relational databases and the object-oriented modeling techniques used in programming languages such as Java, C++, Visual Basic .NET or C#. However, a more popular alternative for achieving such a bridge is to use a standard relational database systems with some form of Object-relational mapping (ORM) software. Whereas traditional RDBMS or SQL-DBMS products focused on the efficient management of data drawn from a limited set of data-types (defined by the relevant language standards), an object-relational DBMS allows software developers to integrate their own types and the methods that apply to them into the DBMS.

The ORDBMS (like ODBMS or OODBMS) is integrated with an object-oriented programming language. The characteristic properties of ORDBMS are 1) complex data, 2) type inheritance, and 3) object behavior. **Complex data** creation in most SQL ORDBMSs is based on preliminary schema definition via the user-defined type (UDT). Hierarchy within structured complex data offers an additional property, **type inheritance**. That is, a structured type can have subtypes that reuse all of its attributes and contain additional attributes specific to the subtype. Another advantage, the **object behavior**, is related with access to the program objects. Such program objects have to be storable and transportable for database processing, therefore they usually are named as persistent objects. Inside a database, all the relations with a persistent program object are relations with its object identifier (OID). All of these points can be addressed in a proper relational system, although the SQL standard and its implementations impose arbitrary restrictions and additional complexityWikipedia:Citing sources

In object-oriented programming (OOP) object behavior is described through the methods (object functions). The methods denoted by one name are distinguished by the type of their parameters and type of objects for which they attached (method signature). The OOP languages call this the polymorphism principle, which briefly is defined as

"one interface, many implementations". Other OOP principles, inheritance and encapsulation are related both, with methods and attributes. Method inheritance is included in type inheritance. Encapsulation in OOP is a visibility degree declared, for example, through the PUBLIC, PRIVATE and PROTECTED modifiers.

History

Object-relational database management systems grew out of research that occurred in the early 1990s. That research extended existing relational database concepts by adding object concepts. The researchers aimed to retain a declarative query-language based on predicate calculus as a central component of the architecture. Probably the most notable research project, Postgres (UC Berkeley), spawned two products tracing their lineage to that research: Illustra and PostgreSQL.

In the mid-1990s, early commercial products appeared. These included Illustra[1] (Illustra Information Systems, acquired by Informix Software which was in turn acquired by IBM), Omniscience (Omniscience Corporation, acquired by Oracle Corporation and became the original Oracle Lite), and UniSQL (UniSQL, Inc., acquired by KCOMS). Ukrainian developer Ruslan Zasukhin, founder of Paradigma Software, Inc., developed and shipped the first version of Valentina database in the mid-1990s as a C++ SDK. By the next decade, PostgreSQL had become a commercially viable database and is the basis for several products today which maintain its ORDBMS features.

Computer scientists came to refer to these products as "object-relational database management systems" or ORDBMSs.[2]

Many of the ideas of early object-relational database efforts have largely become incorporated into SQL:1999 via structured types. In fact, any product that adheres to the object-oriented aspects of SQL:1999 could be described as an object-relational database management product. For example, IBM's DB2, Oracle database, and Microsoft SQL Server, make claims to support this technology and do so with varying degrees of success.

Comparison to RDBMS

An RDBMS might commonly involve SQL statements such as these:

```
CREATE TABLE Customers  (
    Id          CHAR(12)     NOT NULL PRIMARY KEY,
    Surname     VARCHAR(32) NOT NULL,
    FirstName   VARCHAR(32) NOT NULL,
    DOB         DATE         NOT NULL
 );
 SELECT InitCap(Surname) || ', ' || InitCap(FirstName)
   FROM Customers
  WHERE Month(DOB) = Month(getdate())
    AND Day(DOB) = Day(getdate())
```

Most current[3] SQL databases allow the crafting of custom functions, which would allow the query to appear as:

```
SELECT Formal(Id)
  FROM Customers
 WHERE Birthday(DOB) = Today()
```

In an object-relational database, one might see something like this, with user-defined data-types and expressions such as `BirthDay()`:

```
CREATE TABLE Customers (
  Id          Cust_Id     NOT NULL  PRIMARY KEY,
```

```
    Name          PersonName  NOT NULL,
    DOB           DATE        NOT NULL
  );
  SELECT Formal( C.Id )
    FROM Customers C
   WHERE BirthDay ( C.DOB ) = TODAY;
```

The object-relational model can offer another advantage in that the database can make use of the relationships between data to easily collect related records. In an address book application, an additional table would be added to the ones above to hold zero or more addresses for each customer. Using a traditional RDBMS, collecting information for both the user and their address requires a "join":

```
   SELECT InitCap(C.Surname) || ', ' || InitCap(C.FirstName), A.city
     FROM Customers C join Addresses A ON A.Cust_Id=C.Id -- the join
    WHERE A.city="New York"
```

The same query in an object-relational database appears more simply:

```
  SELECT Formal( C.Name )
    FROM Customers C
   WHERE C.address.city="New York" -- the linkage is 'understood' by
the ORDB
```

References

[1] Stonebraker,. Michael with Moore, Dorothy. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers, 1996. ISBN 1-55860-397-2.

[2] There was, at the time, a dispute whether the term was coined by Michael Stonebraker of Illustra or Won Kim of UniSQL.

[3] http://en.wikipedia.org/w/index.php?title=Object-relational_database&action=edit

External links

- Savushkin, Sergey (2003), *A Point of View on ORDBMS* (http://savtechno.com/articles/ViewOfORDBMS.html), retrieved 2012-07-21.
- *JPA Performance Benchmark* (http://www.jpab.org/) – comparison of Java JPA ORM Products (Hibernate, EclipseLink, OpenJPA, DataNucleus).
- *PolePosition Benchmark* (http://www.polepos.org/) – shows the performance trade-offs for solutions in the object-relational impedance mismatch context.

Transaction processing

In computer science, **transaction processing** is information processing that is divided into individual, indivisible operations, called *transactions.* Each transaction must succeed or fail as a complete unit; it cannot remain in an intermediate state.

Since most, though not necessarily all, transaction processing today is interactive the term is often treated as synonymous with *online transaction processing*.

Description

Transaction processing is designed to maintain a database Integrity (typically a database or some modern filesystems) in a known, consistent state, by ensuring that interdependent operations on the system are either all completed successfully or all canceled successfully.

For example, consider a typical banking transaction that involves moving $700 from a customer's savings account to a customer's checking account. This transaction involves at least two separate operations in computer terms: debiting the savings account by $700, and crediting the checking account by $700. If one operation succeeds but the other does not, the books of the bank will not balance at the end of the day. There must therefore be a way to ensure that either both operations succeed or both fail, so that there is never any inconsistency in the bank's database as a whole.

Transaction processing links multiple individual operations in a single, indivisible transaction, and ensures that either all operations in a transaction are completed without error, or none of them are. If some of the operations are completed but errors occur when the others are attempted, the transaction-processing system "rolls back" *all* of the operations of the transaction (including the successful ones), thereby erasing all traces of the transaction and restoring the system to the consistent, known state that it was in before processing of the transaction began. If all operations of a transaction are completed successfully, the transaction is committed by the system, and all changes to the database are made permanent; the transaction cannot be rolled back once this is done.

Transaction processing guards against hardware and software errors that might leave a transaction partially completed. If the computer system crashes in the middle of a transaction, the transaction processing system guarantees that all operations in any uncommitted transactions are cancelled.

Generally, transactions are issued concurrently. If they overlap (i.e. need to touch the same portion of the database), this can create conflicts. For example, if the customer mentioned in the example above has $150 in his savings account and attempts to transfer $100 to a different person while at the same time moving $100 to the checking account, only one of them can succeed. However, forcing transactions to be processed sequentially is inefficient. Therefore, concurrent implementations of transaction processing is programmed to guarantee that the end result reflects a conflict-free outcome, the same as could be reached if executing the transactions sequentially in any order (a property called serializability). In our example, this means that no matter which transaction was issued first, either the transfer to a different person or the move to the checking account succeeds, while the other one fails.

Methodology

The basic principles of all transaction-processing systems are the same. However, the terminology may vary from one transaction-processing system to another, and the terms used below are not necessarily universal.

Rollback

Transaction-processing systems ensure database integrity by recording intermediate states of the database as it is modified, then using these records to restore the database to a known state if a transaction cannot be committed. For example, copies of information on the database *prior* to its modification by a transaction are set aside by the system before the transaction can make any modifications (this is sometimes called a *before image*). If any part of the transaction fails before it is committed, these copies are used to restore the database to the state it was in before the transaction began.

Rollforward

It is also possible to keep a separate journal of all modifications to a database (sometimes called *after images*). This is not required for rollback of failed transactions but it is useful for updating the database in the event of a database failure, so some transaction-processing systems provide it. If the database fails entirely, it must be restored from the most recent back-up. The back-up will not reflect transactions committed since the back-up was made. However, once the database is restored, the journal of after images can be applied to the database (*rollforward*) to bring the database up to date. Any transactions in progress at the time of the failure can then be rolled back. The result is a database in a consistent, known state that includes the results of all transactions committed up to the moment of failure.

Deadlocks

In some cases, two transactions may, in the course of their processing, attempt to access the same portion of a database at the same time, in a way that prevents them from proceeding. For example, transaction A may access portion X of the database, and transaction B may access portion Y of the database. If, at that point, transaction A then tries to access portion Y of the database while transaction B tries to access portion X, a *deadlock* occurs, and neither transaction can move forward. Transaction-processing systems are designed to detect these deadlocks when they occur. Typically both transactions will be cancelled and rolled back, and then they will be started again in a different order, automatically, so that the deadlock doesn't occur again. Or sometimes, just one of the deadlocked transactions will be cancelled, rolled back, and automatically restarted after a short delay.

Deadlocks can also occur between three or more transactions. The more transactions involved, the more difficult they are to detect, to the point that transaction processing systems find there is a practical limit to the deadlocks they can detect.

Compensating transaction

In systems where commit and rollback mechanisms are not available or undesirable, a compensating transaction is often used to undo failed transactions and restore the system to a previous state.

ACID criteria

Jim Gray defined properties of a reliable transaction system in the late 1970s under the acronym *ACID ƒ* atomicity, consistency, isolation, and durability.

Atomicity

A transaction,s changes to the state are atomic: either all happen or none happen. These changes include database changes, messages, and actions on transducers.

Consistency

Consistency: A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state.

Isolation

Even though transactions execute concurrently, it appears to each transaction T, that others executed either before T or after T, but not both.

Durability

Once a transaction completes successfully (commits), its changes to the state survive failures.

Benefits

Transaction processing has these benefits:

- It allows sharing of computer resources among many users
- It shifts the time of job processing to when the computing resources are less busy
- It avoids idling the computing resources without minute-by-minute human interaction and supervision
- It is used on expensive classes of computers to help amortize the cost by keeping high rates of utilization of those expensive resources

Implementations

Standard transaction-processing software, notably IBM's Information Management System, was first developed in the 1960s, and was often closely coupled to particular database management systems. Client€server computing implemented similar principles in the 1980s with mixed success. However, in more recent years, the distributed client€server model has become considerably more difficult to maintain. As the number of transactions grew in response to various online services (especially the Web), a single distributed database was not a practical solution. In addition, most online systems consist of a whole suite of programs operating together, as opposed to a strict client€server model where the single server could handle the transaction processing. Today a number of transaction processing systems are available that work at the inter-program level and which scale to large systems, including mainframes.

One well-known[*citation needed*] (and open) industry standard is the X/Open Distributed Transaction Processing (DTP) (see also JTA the Java Transaction API). However, proprietary transaction-processing environments such as IBM's CICS are still very popular[*citation needed*], although CICS has evolved to include open industry standards as well.

The term 'Extreme Transaction Processing' (XTP) has been used to describe transaction processing systems with uncommonly challenging requirements, particularly throughput requirements (transactions per second). Such systems may be implemented via distributed or cluster style architectures.

References

External references

- Nuts and Bolts of Transaction Processing (http://www.subbu.org/articles/nuts-and-bolts-of-transaction-processing)
- Managing Transaction Processing for SQL Database Integrity (http://www.informit.com/articles/article.aspx?p=174375)

Further reading

- Gerhard Weikum, Gottfried Vossen, *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*, Morgan Kaufmann, 2002, ISBN 1-55860-508-8
- Jim Gray, Andreas Reuter, Transaction Processing: Concepts and Techniques, 1993, Morgan Kaufmann, ISBN 1-55860-190-2
- Philip A. Bernstein, Eric Newcomer, Principles of Transaction Processing, 1997, Morgan Kaufmann, ISBN 1-55860-415-4
- Ahmed K. Elmagarmid (Editor), Transaction Models for Advanced Database Applications, Morgan-Kaufmann, 1992, ISBN 1-55860-214-3

Concepts

ACID

In computer science, **ACID** (*Atomicity, Consistency, Isolation, Durability*) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction. The chosen initials refer to the acid test.[*citation needed*]

Jim Gray defined these properties of a reliable transaction system in the late 1970s and developed technologies to achieve them automatically.[1]

In 1983, Andreas Reuter and Theo HŽrder coined the acronym *ACID* to describe them.[2]

Characteristics

Atomicity

Atomicity requires that each transaction is "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible ("atomic"), and an aborted transaction does not happen.

Consistency

The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including but not limited to constraints, cascades, triggers, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors do not violate any defined rules.

Isolation

The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e. one after the other. Providing isolation is the main goal of concurrency control. Depending on concurrency control method, the effects of an incomplete transaction might not even be visible to another transaction.[*citation needed*]

Durability

Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory.

Examples

The following examples further illustrate the ACID properties. In these examples, the database table has two columns, A and B. An integrity constraint requires that the value in A and the value in B must sum to 100. The following SQL code creates a table as described above:

```
CREATE TABLE acidtest (A INTEGER, B INTEGER CHECK (A + B = 100));
```

Atomicity failure

Assume that a transaction attempts to subtract 10 from A and add 10 to B. This is a valid transaction, since the data continue to satisfy the constraint after it has executed. However, assume that after removing 10 from A, the transaction is unable to modify B. If the database retained A's new value, atomicity and the constraint would both be violated. Atomicity requires that both parts of this transaction, or neither, be complete.

Consistency failure

Consistency is a very general term which demands that the data must meet all validation rules. In the previous example, the validation is a requirement that A + B = 100. Also, it may be inferred that both A and B must be integers. A valid range for A and B may also be inferred. All validation rules must be checked to ensure consistency.

Assume that a transaction attempts to subtract 10 from A without altering B. Because consistency is checked after each transaction, it is known that A + B = 100 before the transaction begins. If the transaction removes 10 from A successfully, atomicity will be achieved. However, a validation check will show that A + B = 90, which is inconsistent with the rules of the database. The entire transaction must be cancelled and the affected rows rolled back to their pre-transaction state. If there had been other constraints, triggers, or cascades, every single change operation would have been checked in the same way as above before the transaction was committed.

Isolation failure

To demonstrate isolation, we assume two transactions execute at the same time, each attempting to modify the same data. One of the two must wait until the other completes in order to maintain isolation.

Consider two transactions. T_1 transfers 10 from A to B. T_2 transfers 10 from B to A. Combined, there are four actions:

- T_1 subtracts 10 from A.
- T_1 adds 10 to B.
- T_2 subtracts 10 from B.
- T_2 adds 10 to A.

If these operations are performed in order, isolation is maintained, although T_2 must wait. Consider what happens if T_1 fails half-way through. The database eliminates T_1's effects, and T_2 sees only valid data.

By interleaving the transactions, the actual order of actions might be:

- T_1 subtracts 10 from A.
- T_2 subtracts 10 from B.
- T_2 adds 10 to A.
- T_1 adds 10 to B.

Again, consider what happens if T_1 fails halfway through. By the time T_1 fails, T_2 has already modified A; it cannot be restored to the value it had before T_1 without leaving an invalid database. This is known as a write-write failure,[*citation needed*] because two transactions attempted to write to the same data field. In a typical system, the problem would be resolved by reverting to the last known good state, canceling the failed transaction T_1, and restarting the interrupted transaction T_2 from the good state.

Durability failure

Assume that a transaction transfers 10 from A to B. It removes 10 from A. It then adds 10 to B. At this point, a "success" message is sent to the user. However, the changes are still queued in the disk buffer waiting to be committed to the disk. Power fails and the changes are lost. The user assumes (understandably) that the changes have been made.

Implementation

Processing a transaction often requires a sequence of operations that is subject to failure for a number of reasons. For instance, the system may have no room left on its disk drives, or it may have used up its allocated CPU time.

There are two popular families of techniques: write ahead logging and shadow paging. In both cases, locks must be acquired on all information that is updated, and depending on the level of isolation, possibly on all data that is read as well. In write ahead logging, atomicity is guaranteed by copying the original (unchanged) data to a log before changing the database.Wikipedia:Disputed statement That allows the database to return to a consistent state in the event of a crash.

In shadowing, updates are applied to a partial copy of the database, and the new copy is activated when the transaction commits.

Locking vs multiversioning

Many databases rely upon locking to provide ACID capabilities. Locking means that the transaction marks the data that it accesses so that the DBMS knows not to allow other transactions to modify it until the first transaction succeeds or fails. The lock must always be acquired before processing data, including data that are read but not modified. Non-trivial transactions typically require a large number of locks, resulting in substantial overhead as well as blocking other transactions. For example, if user A is running a transaction that has to read a row of data that user B wants to modify, user B must wait until user A's transaction completes. Two phase locking is often applied to guarantee full isolation.[*citation needed*]

An alternative to locking is multiversion concurrency control, in which the database provides each reading transaction the prior, unmodified version of data that is being modified by another active transaction. This allows readers to operate without acquiring locks, i.e. writing transactions do not block reading transactions, and readers do not block writers. Going back to the example, when user A's transaction requests data that user B is modifying, the database provides A with the version of that data that existed when user B started his transaction. User A gets a consistent view of the database even if other users are changing data. One implementation, namely snapshot isolation, relaxes the isolation property.

Distributed transactions

Guaranteeing ACID properties in a distributed transaction across a distributed database where no single node is responsible for all data affecting a transaction presents additional complications. Network connections might fail, or one node might successfully complete its part of the transaction and then be required to roll back its changes, because of a failure on another node. The two-phase commit protocol (not to be confused with two-phase locking) provides atomicity for distributed transactions to ensure that each participant in the transaction agrees on whether the transaction should be committed or not.[*citation needed*] Briefly, in the first phase, one node (the coordinator) interrogates the other nodes (the participants) and only when all reply that they are prepared does the coordinator, in the second phase, formalize the transaction.

References

[1] Gray, Jim, and Reuter, Andreas, *Distributed Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. ISBN 1-55860-190-2.

[2] These four properties, atomicity, consistency, isolation, and durability (ACID), describe the major highlights of the transaction paradigm, which has influenced many aspects of development in database systems.

Create, read, update and delete

In computer programming, **create, read, update and delete** (**CRUD**) (Sometimes called SCRUD with an "S" for Search) are the four basic functions of persistent storage. Sometimes *CRUD* is expanded with the words *retrieve* instead of *read*, *modify* instead of *update*, or *destroy* instead of *delete*. It is also sometimes used to describe user interface conventions that facilitate viewing, searching, and changing information; often using computer-based forms and reports. The term was likely first popularized by James Martin in his 1983 book *Managing the Data-base Environment*. The acronym may be extended to CRUDL to cover *listing* of large data sets which bring additional complexity such as pagination when the data sets are too large to hold easily in memory.

Another variation of CRUD is BREAD, an acronym for "Browse, Read, Edit, Add, Delete".

Database applications

The acronym CRUD refers to all of the major functions that are implemented in relational database applications. Each letter in the acronym can map to a standard SQL statement and HTTP method:

Operation	SQL	HTTP
Create	INSERT	POST
Read (Retrieve)	SELECT	GET
Update (Modify)	UPDATE	PUT / PATCH
Delete (Destroy)	DELETE	DELETE

Making full use of HTTP methods, along with other constraints, is considered "RESTful".

Although a relational database provides a common persistence layer in software applications, numerous other persistence layers exist. CRUD functionality can be implemented with an object database, an XML database, flat text files, custom file formats, tape, or card, for example.

User interface

CRUD is also relevant at the user interface level of most applications. For example, in address book software, the basic storage unit is an individual *contact entry*. As a bare minimum, the software must allow the user to:

- Create or add new entries
- Read, retrieve, search, or view existing entries
- Update or edit existing entries
- Delete/deactivate existing entries

Without at least these four operations, the software cannot be considered complete. Because these operations are so fundamental, they are often documented and described under one comprehensive heading, such as "contact management", "content management" or "contact maintenance" (or "document management" in general, depending on the basic storage unit for the particular application).

Notes

Null (SQL)

Null is a special marker used in Structured Query Language (SQL) to indicate that a data value does not exist in the database. Introduced by the creator of the relational database model, E. F. Codd, SQL Null serves to fulfill the requirement that all *true relational database management systems (RDBMS)* support a representation of "missing information and inapplicable information". Codd also introduced the use of the lowercase Greek omega (Œ) symbol to represent Null in database theory. NULL is also an SQL reserved keyword used to identify the Null special marker.

The Greek lowercase omega (Œ) character is used to represent Null in database theory.

Null has been the focus of controversy and a source of debate because of its associated three-valued logic (3VL), special requirements for its use in SQL joins, and the special handling required by aggregate functions and SQL grouping operators. Computer science professor Ron van der Meyden summarized the various issues as: "The inconsistencies in the SQL standard mean that it is not possible to ascribe any intuitive logical semantics to the treatment of nulls in SQL."[1] Although various proposals have been made for resolving these issues, the complexity of the alternatives has prevented their widespread adoption.

For people new to the subject, a good way to remember what null means is to remember that in terms of information, "lack of a value" is not the same thing as "a value of zero"; similarly, "lack of an answer" is not the same thing as "an answer of no". For example, consider the question "How many books does Juan own?" The answer may be "zero" (we *know* that he owns *none*) or "null" (we *do not know* how many he owns, or doesn't own). In a database table, the column reporting this answer would start out with a value of null, and it would not be updated with "zero" until we have ascertained that Juan owns no books.

History

E. F. Codd mentioned nulls as a method of representing missing data in the relational model in an 1975 paper in the *FDT Bulletin of ACM-SIGMOD*. Codd's paper that is most commonly cited in relation with the semantics of Null (as adopted in SQL) is his 1979 paper in the *ACM Transactions on Database Systems*, in which he also introduced his Relational Model/Tasmania, although much of the other proposals from the latter paper have remained obscure. Section 2.3 of his 1979 paper details the semantics of Null propagation in arithmetic operations and well as comparisons employing a ternary (three-valued) logic when comparing to nulls; it also details the treatment of Nulls on other set operations (the latter issue still controversial today). In database theory circles, the original proposal of Codd (1975, 1979) is now referred to as "Krokk tables". Codd later reinforced his requirement that all RDBMS support Null to indicate missing data in a 1985 two-part article published in *ComputerWorld* magazine.

The 1986 SQL standard basically adopted Codd's proposal after an implementation prototype in IBM System R. Although Don Chamberlin recognized nulls (alongside duplicate rows) as one of the most controversial features of SQL, he defended the design of Nulls in SQL invoking the pragmatic arguments that it was the least expensive form of system support for missing information, saving the programmer from many duplicative application-level checks (see semipredicate problem) while at the same time providing the database designer with the option not to use nulls if he so desires; for example, in order to avoid well known anomalies (discussed in the semantics section of this article). Chamberlin also argued that besides providing some missing-value functionality, practical experience with Nulls also led to other language features which rely on Nulls, like certain grouping constructs and outer joins.

Finally, he argued that in practice Nulls also end up being used a quick way to patch an existing schema when it needs to evolve beyond its original intent, coding not for missing but rather for inapplicable information; for example, a database that quickly needs to support electric cars while having a miles-per-gallon column.

Codd indicated in his 1990 book *The Relational Model for Database Management, Version 2* that the single Null mandated by the SQL standard was inadequate, and should be replaced by two separate Null-type markers to indicate the reason why data is missing. In Codd's book, these two Null-type markers are referred to as 'A-Values' and 'I-Values', representing 'Missing But Applicable' and 'Missing But Inapplicable', respectively. Codd's recommendation would have required SQL's logic system be expanded to accommodate a four-valued logic system. Because of this additional complexity, the idea of multiple Null-type values has not gained widespread acceptance in the database practitioners' domain. It remains an active field of research though, with numerous papers still being published.

Null propagation

Arithmetic operations

Because Null is not a data value, but a marker for an unknown value, using mathematical operators on Null results in an unknown value, which is represented by Null. In the following example, multiplying 10 by Null results in Null:

```
10 * NULL          -- Result is NULL
```

This can lead to unanticipated results. For instance, when an attempt is made to divide Null by zero, platforms may return Null instead of throwing an expected "data exception - division by zero". Though this behavior is not defined by the ISO SQL standard many DBMS vendors treat this operation similarly. For instance, the Oracle, PostgreSQL, MySQL Server, and Microsoft SQL Server platforms all return a Null result for the following:

```
NULL / 0
```

String concatenation

String concatenation operations, which are common in SQL, also result in Null when one of the operands is Null. The following example demonstrates the Null result returned by using Null with the SQL || string concatenation operator.

```
'Fish ' || NULL || 'Chips'   -- Result is NULL
```

This is not true for all database implementations. In an Oracle RDBMS for example NULL and the empty string are considered the same thing and therefore 'Fish ' || NULL || 'Chips' results in 'Fish Chips'.

Comparisons with NULL and the three-valued logic (3VL)

Since Null is not a member of any data domain, it is not considered a "value", but rather a marker (or placeholder) indicating the absence of value. Because of this, comparisons with Null can never result in either True or False, but always in a third logical result, Unknown. The logical result of the expression below, which compares the value 10 to Null, is Unknown:

```
SELECT 10 = NULL       -- Results in Unknown
```

However, certain operations on Null can return values if the value of Null is not relevant to the outcome of the operation. Consider the following example:

```
SELECT NULL OR TRUE    -- Results in True
```

In this case, the fact that the value on the left of OR is unknowable is irrelevant, because the outcome of the OR operation would be True regardless of the value on the left.

SQL implements three logical results, so SQL implementations must provide for a specialized three-valued logic (3VL). The rules governing SQL three-valued logic are shown in the tables below (**p** and **q** represent logical states)" The truth tables SQL uses for AND, OR, and NOT correspond to a common fragment of the Kleene and •ukasiewicz three-valued logic (which differ in their definition of implication, however SQL defines no such operation).

p	*q*	*p* OR *q*	*p* AND *q*	*p* = *q*
True	True	True	True	True
True	False	True	False	False
True	Unknown	True	Unknown	Unknown
False	True	True	False	False
False	False	False	False	True
False	Unknown	Unknown	False	Unknown
Unknown	True	True	Unknown	Unknown
Unknown	False	Unknown	False	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown

p	NOT *p*
True	False
False	True
Unknown	Unknown

Effect of Unknown in WHERE clauses

SQL three-valued logic is encountered in Data Manipulation Language (DML) in comparison predicates of DML statements and queries. The WHERE clause causes the DML statement to act on only those rows for which the predicate evaluates to True. Rows for which the predicate evaluates to either False or Unknown are not acted on by INSERT, UPDATE, or DELETE DML statements, and are discarded by SELECT queries. Interpreting Unknown and False as the same logical result is a common error encountered while dealing with Nulls. The following simple example demonstrates this fallacy:

```
SELECT *
FROM t
WHERE i = NULL;
```

The example query above logically always returns zero rows because the comparison of the *i* column with Null always returns Unknown, even for those rows where *i* is Null. The Unknown result causes the SELECT statement to summarily discard each and every row. (However, in practice, some SQL tools will retrieve rows using a comparison with Null.)

Null-specific and 3VL-specific comparison predicates

Basic SQL comparison operators always return Unknown when comparing anything with Null, so the SQL standard provides for two special Null-specific comparison predicates. The `IS NULL` and `IS NOT NULL` predicates (which use a postfix syntax) test whether data is, or is not, Null.

The SQL standard contains an extension F571 "Truth value tests" that introduces three additional logical unary operators (six in fact, if we count their negation, which is part of their syntax), also using postfix notation. They have the following truth tables:[2]

p	true	false	unknown
p IS TRUE	true	false	false
p IS NOT TRUE	false	true	true
p IS FALSE	false	true	false
p IS NOT FALSE	true	false	true
p IS UNKNOWN	false	false	true
p IS NOT UNKNOWN	true	true	false

The F571 extension is orthogonal to the presence of the boolean datatype in SQL (discussed later in this article) and, despite syntactic similarities, F571 does not introduce boolean or three-valued literals in the language. The F571 extension was actually present in SQL92, well before the boolean datatype was introduced to the standard in 1999. The F571 extension is implemented by few systems however; PostgreSQL is one of those implementing it.

The addition of IS UNKNOWN to the other operators of SQL's three-valued logic makes the SQL three-valued logic functionally complete,[3] meaning its logical operators can express (in combination) any conceivable three-valued logical function.

On systems which don't support the F571 extension, it is possible to emulate IS UNKNOWN *p* by going over every argument that could make the expression *p* Unknown and test those arguments with IS NULL or other NULL-specific functions, although this may be more cumbersome.

Law of the excluded fourth (in WHERE clauses)

In SQL's three-valued logic the law of the excluded middle, *p* OR NOT *p*, no longer evaluates to true for all *p*. More precisely, in SQL's three-valued logic *p* OR NOT *p* is unknown precisely when *p* is unknown and true otherwise. Because direct comparisons with Null result in the unknown logical value, the following query

```
SELECT * FROM stuff WHERE ( x = 10 ) OR NOT ( x = 10 );
```

is not equivalent in SQL with

```
SELECT * FROM stuff;
```

if the column x contains any Nulls; in that case the second query would return some rows the first one does not return, namely all those in which x is Null. In classical two-valued logic, the law of the excluded middle would allow the simplification of the WHERE clause predicate, in fact its elimination. Attempting to apply the law of the excluded middle to SQL's 3VL is effectively a false dichotomy. The second query is actually equivalent with:

```
SELECT * FROM stuff;
-- is (because of 3VL) equivalent to:
SELECT * FROM stuff WHERE ( x = 10 ) OR NOT ( x = 10 ) OR x IS NULL;
```

Thus, to correctly simplify the first statement in SQL requires that we return all rows in which x is not null.

```
SELECT * FROM stuff WHERE x IS NOT NULL;
```

From the above, it's easy observe that for SQL's WHERE clause a tautology similar to the law of excluded middle can be written. Assuming the IS UNKNOWN operator is present, this is *p* OR (NOT *p*) OR (*p* IS UNKNOWN) is true for every predicate *p*. Among logicians, this is called law of excluded fourth.

There are some SQL expressions in which it is less obvious where the false dilemma occurs, for example:

```
SELECT 'ok' WHERE 1 NOT IN (SELECT CAST (NULL AS INTEGER))
UNION
SELECT 'ok' WHERE 1 IN (SELECT CAST (NULL AS INTEGER));
```

produces no rows because `IN` is translates to an iterated version of equality over the argument set and 1<>NULL is Unknown, just a as 1=NULL is Unknown. (The CAST in this example is needed only in some SQL implementations like PostgreSQL, which would reject it with a type checking error otherwise. In many systems plain SELECT NULL works in the subquery.) The missing case above is of course:

```
SELECT 'ok' WHERE (1 IN (SELECT CAST (NULL AS INTEGER))) IS UNKNOWN;
```

Effect of Null and Unknown in other constructs

Joins

Joins evaluate using the same comparison rules as for WHERE clauses. Therefore, care must be taken when using nullable columns in SQL join criteria. In particular a table containing any nulls is *not equal* with a natural self-join of itself, meaning that whereas $R \bowtie R = R$ is true for any relation *R* in relational algebra, a SQL self-join will exclude all rows having a null value anywhere.[4] An example of this behavior is given in the section analyzing the missing-value semantics of Nulls.

The SQL `COALESCE` function or `CASE` expressions can be used to "simulate" Null equality in join criteria, and the `IS NULL` and `IS NOT NULL` predicates can be used in the join criteria as well. The following predicate tests for equality of the values A and B and treats Nulls as being equal.

```
(A = B) OR (A IS NULL AND B IS NULL)
```

CASE expressions

SQL provides two flavours of conditional expressions. One is called "simple CASE" and operates like a switch statement. The other is called a "searched CASE" in the standard, and operates like an if...elseif.

The simple `CASE` expressions use implicit equality comparisons which operate under the same rules as the DML `WHERE` clause rules for Null. Thus, a *simple CASE expression* cannot check for the existence of Null directly. A check for Null in a simple `CASE` expression always results in Unknown, as in the following:

```
SELECT CASE i WHEN NULL THEN 'Is Null'  -- This will never be returned
              WHEN    0 THEN 'Is Zero'  -- This will be returned when i
 = 0
              WHEN    1 THEN 'Is One'   -- This will be returned when i
 = 1
              END
FROM t;
```

Because the expression `i = NULL` evaluates to Unknown no matter what value column *i* contains (even if it contains Null), the string `'Is Null'` will never be returned.

On the other hand, a "searched" CASE expression can use predicates like IS NULL and IS NOT NULL in its conditions. The following example shows how to use a searched CASE expression to properly check for Null:

```
SELECT CASE WHEN i IS NULL THEN 'Null Result'  -- This will be returned
 when i is NULL
            WHEN      i = 0 THEN 'Zero'         -- This will be returned
 when i = 0
            WHEN      i = 1 THEN 'One'          -- This will be returned
 when i = 1
            END
FROM t;
```

In the searched CASE expression, the string 'Null Result' is returned for all rows in which *i* is Null.

Oracle's dialect of SQL provides a built-in function DECODE which can be used instead of the simple CASE expressions and considers two nulls equal.

```
SELECT DECODE(i, NULL, 'Null Result', 0, 'Zero', 1, 'One') FROM t;
```

Finally, all these constructs return a NULL if no match is found; they have a default ELSE NULL clause.

IF statements in procedural extensions

SQL/PSM (SQL Persistent Stored Modules) defines procedural extensions for SQL, such as the IF statement. However, the major SQL vendors have historically included their own proprietary procedural extensions. Procedural extensions for looping and comparisons operate under Null comparison rules similar to those for DML statements and queries. The following code fragment, in ISO SQL standard format, demonstrates the use of Null 3VL in an IF statement.

```
IF i = NULL THEN
      SELECT 'Result is True'
ELSEIF NOT(i = NULL) THEN
      SELECT 'Result is False'
ELSE
      SELECT 'Result is Unknown';
```

The IF statement performs actions only for those comparisons that evaluate to True. For statements that evaluate to False or Unknown, the IF statement passes control to the ELSEIF clause, and finally to the ELSE clause. The result of the code above will always be the message 'Result is Unknown' since the comparisons with Null always evaluate to Unknown.

Analysis of SQL Null missing-value semantics

The groundbreaking work of T. Imielinski and W. Lipski (1984) provided a framework in which to evaluate the intended semantics of various proposals to implement missing-value semantics. This section roughly follows chapter 19 the "Alice" textbook. A similar presentation appears in the review of Ron van der Meyden, •10.4.

In selections and projections: weak representation

Constructs representing missing information, such as Codd tables, are actually intended to represent a set of relations, one for each possible instantiation of their parameters; in the case of Codd tables, this means replacement of Nulls with some concrete value. For example,

the Codd table may represent the relation or equally well the relation

Name	Age
George	43
Harriet	NULL
Charles	56

Name	Age
George	43
Harriet	22
Charles	56

Name	Age
George	43
Harriet	37
Charles	56

A construct (such as a Codd table) is said to be a *strong representation* system (of missing information) if any answer to a query made on the construct can be particularized to obtain an answer for *any* corresponding query on the relations it represents, which are seen as models of the construct. More precisely, if q is a query formula in the relational algebra (of "pure" relations) and if $\bar{q}$ is its lifting to a construct intended to represent missing information, a strong representation has the property that for any query *q* and (table) construct *T*, $\bar{q}$ lifts *all* the answers to the construct, i.e.:

$$\mathrm{Models}(\bar{q}(T)) = \{q(R) \,|R \in \mathrm{Models}(T)\}$$

(The above has to hold for queries taking any number of tables as arguments, but the restriction to one table suffices for this discussion.) Clearly Codd tables do not have this strong property if selections and projections are considered as part of the query language. For example, *all* the answers to

```
SELECT * FROM Emp WHERE Age = 22;
```

should include the possibility that a relation like EmpH22 may exist. However Codd tables cannot represent the disjunction "result with possibly 0 or 1 rows". A device, mostly of theoretical interest, called conditional table (or c-table) can however represent such an answer:

Result

Name	Age	condition
Harriet	$\mathbb{E}_1$	$\mathbb{E}_1 = 22$

where the condition column is interpreted as the row doesn't exist if the condition is false. It turns out that because the formulas in the condition column of a c-table can be arbitrary propositional logic formulas, an algorithm for the problem whether a c-table represents some concrete relation has a co-NP-complete complexity, thus is of little practical value.

A weaker notion of representation is therefore desirable. Imielinski and Lipski introduced the notion of *weak representation*, which essentially allows (lifted) queries over a construct to return a representation only for *sure* information, i.e. if it's valid for all "possible world" instantiations (models) of the construct. Concretely, a construct is a weak representation system if

$$\bigcap \mathrm{Models}(\bar{q}(T)) = \bigcap \{q(R) \,|R \in \mathrm{Models}(T)\}$$

The right-hand side of the above equation is the *sure* information, i.e. information which can be certainly extracted from the database regardless of what values are used to replace Nulls in the database. In the example we considered above, it's easy to see that the intersection of all possible models (i.e. the sure information) of the query selecting WHERE Age = 22 is actually empty because, for instance, the (unlifted) query returns no rows for the relation EmpH37. More generally, it was shown by Imielinski and Lipski that Codd tables are a weak representation system if the query language is restricted to projections, selections (and renaming of columns). However, as soon as we add either joins or unions to the query language, even this weak property is lost, as evidenced in the next section.

If joins or unions are considered: not even weak representation

Let us consider the following query over the same Codd table Emp from the previous section:

```
SELECT Name FROM Emp WHERE Age = 22
UNION
SELECT Name FROM Emp WHERE Age <> 22;
```

Whatever concrete value one would choose for the NULL age of Harriet, the above query will return the full column of names of any model of Emp, but when the (lifted) query is ran on Emp itself, Harriet will always be missing, i.e. we have:

Query result on Emp:

Name
George
Charles

Query result on any model of Emp:

Name
George
Harriet
Charles

Thus when unions are added to the query language, Codd tables are not even a weak representation system of missing information, meaning that queries over them don't even report all *sure* information. It's important to note here that semantics of UNION on Nulls, which are discussed in a later section, did not even come into play in this query. The "forgetful" nature of the two sub-queries was all that it took to guarantee that some sure information went unreported when the above query was ran on the Codd table Emp.

For natural joins, the example needed to show that sure information may be unreported by some query is slightly more complicated. Consider the table

J

F1	F2	F3
11	NULL	13
21	NULL	23
31	32	33

and the query

```
SELECT F1, F3 FROM
  (SELECT F1, F2 FROM J) AS F12
  NATURAL JOIN
  (SELECT F2, F3 FROM J) AS F23;
```

Query result on J:

F1	F3
31	33

Query result on any model of J:

F1	F3
11	13
21	23
31	33

The intuition for what happens above is that the Codd tables representing the projections in the subqueries lose track of the fact that the Null values in the columns F12.F2 and F23.F2 are actually copies of the originals in the table J.

This observation suggests that a relatively simple improvement of Codd tables (which works correctly for this example) would be to use *Skolem constants* (meaning Skolem functions which are also constant functions), say $Œ_{12}$ and $Œ_{22}$ instead of a single NULL symbol. Such an approach, called v-tables or Naive tables, is computationally less expensive that the c-tables discussed above. However it is still not a complete solution for incomplete information in the sense that v-tables are only a weak representation for queries not using any negations in selection (and not using any set difference either). The first example considered in this section is using a negative selection clause, WHERE Age <> 22, so it is also an example where v-tables queries would not report sure information.

Check constraints and foreign keys

The primary place in which SQL three-valued logic intersects with SQL Data Definition Language (DDL) is in the form of check constraints. A check constraint placed on a column operates under a slightly different set of rules than those for the DML `WHERE` clause. While a DML `WHERE` clause must evaluate to True for a row, a check constraint must not evaluate to False. (From a logic perspective, the designated values are True and Unknown.) This means that a check constraint will succeed if the result of the check is either True or Unknown. The following example table with a check constraint will prohibit any integer values from being inserted into column *i*, but will allow Null to be inserted since the result of the check will always evaluate to Unknown for Nulls.

```
CREATE TABLE t (
      i INTEGER,
      CONSTRAINT ck_i CHECK ( i < 0 AND i = 0 AND i > 0 ) );
```

Because of the change in designated values relative to the WHERE clause, from a logic perspective the law of excluded middle is a tautology for CHECK constraints, meaning CHECK (*p* OR NOT *p*) always succeeds. Furthermore, assuming Nulls are to be interpreted as existing but unknown values, some pathological CHECKs like the one above allow insertion of Nulls that could never be replaced by any non-null value.

In order to constrain a column to reject Nulls, the `NOT NULL` constraint can be applied, as shown in the example below. The `NOT NULL` constraint is semantically equivalent to a check constraint with an `IS NOT NULL` predicate.

```
CREATE TABLE t ( i INTEGER NOT NULL );
```

By default check constraints against foreign keys succeed if any of the fields in such keys are Null. For example the table

```
CREATE TABLE Books
( title VARCHAR(100),
  author_last VARCHAR(20),
  author_first VARCHAR(20),
FOREIGN KEY (author_last, author_first)
  REFERENCES Authors(last_name, first_name));
```

would allow insertion of rows where author_last or author_first are NULL irrespective of how the table Authors is defined or what it contains. More precisely, a null in any of these fields would allow any value in the other one, even on that is not found in Authors table. For example if Authors contained only ('Doe', 'John'), then ('Smith', NULL) would satisfy the foreign key constraint. SQL-92 added two extra options for narrowing down the matches in such cases. If `MATCH PARTIAL` is added after the `REFERENCES` declaration then any non-null must match the foreign key, e. g. ('Doe', NULL) would still match, but ('Smith', NULL) would not. Finally, if `MATCH FULL` is added then ('Smith', NULL) would not match the constraint either, but (NULL, NULL) would still match it.

Outer joins

SQL outer joins, including left outer joins, right outer joins, and full outer joins, automatically produce Nulls as placeholders for missing values in related tables. For left outer joins, for instance, Nulls are produced in place of rows missing from the table appearing on the right-hand side of the `LEFT OUTER JOIN` operator. The following simple example uses two tables to demonstrate Null placeholder production in a left outer join.

The first table (**Employee**) contains employee ID numbers and names, while the second table (**PhoneNumber**) contains related employee ID numbers and phone numbers, as shown below.

ID	LastName	FirstName
1	Johnson	Joe
2	Lewis	Larry
3	Thompson	Thomas
4	Patterson	Patricia

|+ Employee

ID	Number
1	555-2323
3	555-9876

|+ PhoneNumber

The following sample SQL query performs a left outer join on these two tables.

```
SELECT e.ID, e.LastName, e.FirstName, pn.Number
FROM Employee e
LEFT OUTER JOIN PhoneNumber pn
ON e.ID = pn.ID;
```

The result set generated by this query demonstrates how SQL uses Null as a placeholder for values missing from the right-hand (**PhoneNumber**) table, as shown below.

ID	LastName	FirstName	Number
1	Johnson	Joe	555-2323
2	Lewis	Larry	**NULL**
3	Thompson	Thomas	555-9876
4	Patterson	Patricia	**NULL**

|+ Query result

Aggregate functions

SQL defines aggregate functions to simplify server-side aggregate calculations on data. Except for the `COUNT(*)` function, all aggregate functions perform a Null-elimination step, so that Null values are not included in the final result of the calculation.

Note that the elimination of Null values is not equivalent to replacing those values with zero. For example, in the following table, `AVG(i)` (the average of the values of `i`) will give a different result from that of `AVG(j)`:

i	j
150	150
200	200
250	250
NULL	0

|+ Table

Here AVG(i) is 200 (the average of 150, 200, and 250), while AVG(j) is 150 (the average of 150, 200, 250, and 0). A well-known side effect of this is that in SQL AVG(z) is not equivalent with SUM(z) / COUNT(*).

When two nulls are equal: grouping, sorting, and some set operations

Because SQL:2003 defines all Null markers as being unequal to one another, a special definition was required in order to group Nulls together when performing certain operations. SQL defines "any two values that are equal to one another, or any two Nulls", as "not distinct". This definition of *not distinct* allows SQL to group and sort Nulls when the GROUP BY clause (and other keywords that perform grouping) are used.

Other SQL operations, clauses, and keywords use "not distinct" in their treatment of Nulls. These include the following:

- PARTITION BY clause of ranking and windowing functions like ROW_NUMBER
- UNION, INTERSECT, and EXCEPT operator, which treat NULLs as the same for row comparison/elimination purposes
- DISTINCT keyword used in SELECT queries

The principle that Nulls aren't equal to each other (but rather that the result is Unknown) is effectively violated in the SQL specification for the UNION operator, which does identify nulls with each other. Consequently, some set operations in SQL, like union or difference, may produce results not representing sure information, unlike operations involving explicit comparisons with NULL (e.g. those in a WHERE clause discussed above). In Codd's 1979 proposal (which was basically adopted by SQL92) this semantic inconsistency is rationalized by arguing that removal of duplicates in set operations happens "at a lower level of detail than equality testing in the evaluation of retrieval operations."

The SQL standard does not explicitly define a default sort order for Nulls. Instead, on conforming systems, Nulls can be sorted before or after all data values by using the NULLS FIRST or NULLS LAST clauses of the ORDER BY list, respectively. Not all DBMS vendors implement this functionality, however. Vendors who do not implement this functionality may specify different treatments for Null sorting in the DBMS.

Effect on index operation

Some SQL products do not index keys containing NULL values. For instance, PostgreSQL versions prior to 8.3 did not, with the documentation for a B-tree index stating that

> B-trees can handle equality and range queries on data that can be sorted into some ordering. In particular, the PostgreSQL query planner will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators: < Š = ‰ >
>
> Constructs equivalent to combinations of these operators, such as BETWEEN and IN, can also be implemented with a B-tree index search. (But note that IS NULL is not equivalent to = and is not indexable.)

In cases where the index enforces uniqueness, NULL values are excluded from the index and uniqueness is not enforced between NULL values. Again, quoting from the PostgreSQL documentation:

> When an index is declared unique, multiple table rows with equal indexed values will not be allowed. Null values are not considered equal. A multicolumn unique index will only reject cases where all of the indexed columns are equal in two rows.

This is consistent with the SQL:2003-defined behavior of scalar Null comparisons.

Another method of indexing Nulls involves handling them as *not distinct* in accordance with the SQL:2003-defined behavior. For example, Microsoft SQL Server documentation states the following:

> For indexing purposes, NULL values compare as equal. Therefore, a unique index, or UNIQUE constraint, cannot be created if the key values are NULL in more than one row. Select columns that are defined as NOT NULL when columns for a unique index or unique constraint are chosen.

Both of these indexing strategies are consistent with the SQL:2003-defined behavior of Nulls. Because indexing methodologies are not explicitly defined by the SQL:2003 standard, indexing strategies for Nulls are left entirely to the vendors to design and implement.

Null-handling functions

SQL defines two functions to explicitly handle Nulls: NULLIF and COALESCE. Both functions are abbreviations for searched CASE expressions.

NULLIF

The NULLIF function accepts two parameters. If the first parameter is equal to the second parameter, NULLIF returns Null. Otherwise, the value of the first parameter is returned.

```
NULLIF(value1, value2)
```

Thus, NULLIF is an abbreviation for the following CASE expression:

```
CASE WHEN value1 = value2 THEN NULL ELSE value1 END
```

COALESCE

The COALESCE function accepts a list of parameters, returning the first non-Null value from the list:

```
COALESCE(value1, value2, value3, ...)
```

COALESCE is defined as shorthand for the following SQL CASE expression:

```
CASE WHEN value1 IS NOT NULL THEN value1
     WHEN value2 IS NOT NULL THEN value2
     WHEN value3 IS NOT NULL THEN value3
     ...
     END
```

Some SQL DBMSs implement vendor-specific functions similar to COALESCE. Some systems (e.g. Transact-SQL) implement an ISNULL function, or other similar functions that are functionally similar to COALESCE. (See Is functions for more on the IS functions in Transact-SQL.)

NVL

The Oracle NVL function accepts two parameters. It returns the first non-NULL parameter or NULL if all parameters are NULL.

A COALESCE expression can be converted into an equivalent NVL expression thus:

```
COALESCE ( val1, ... , val{n} )
```

turns into:

```
NVL( val1 , NVL( val2 , NVL( val3 , ‘ , NVL ( val{n-1} , val{n} ) ‘ )))
```

A use case of this function is to replace in an expression a NULL value by a fixed value like in NVL(SALARY, 0) which says, 'if SALARY contains a NULL value, replace it with 0'.

There is, however, one notable exception. In most implementations, COALESCE evaluates its parameters until it reaches the first non-NULL one, while NVL evaluates all of its parameters. This is important for several reasons. A parameter *after* the first non-NULL parameter could be a function, which could either be computationally expensive, invalid, or could create unexpected side effects.

Data typing of Null and Unknown

The NULL literal is untyped in SQL, meaning that it is not designated as an integer, character, or any other specific data type. Because of this, it is sometimes mandatory (or desirable) to explicitly convert Nulls to a specific data type. For instance, if overloaded functions are supported by the RDBMS, SQL might not be able to automatically resolve to the correct function without knowing the data types of all parameters, including those for which Null is passed.

Conversion from the NULL literal to a Null of a specific type is possible using the CAST introduced in SQL-92. For example:

```
CAST (NULL AS INTEGER)
```

represents an integer which has the Null value.

The actual typing of Unknown (distinct or not from NULL itself) varies between SQL implementations. For example the following

```
SELECT 'ok' WHERE (NULL <> 1) IS NULL;
```

parses and executes successfully in some environments (e.g. SQLite or PostgreSQL) which unify a NULL boolean with Unknown but fails to parse in others (e.g. in SQL Server Compact). MySQL behaves similarly to PostgreSQL in this regard (with the minor exception that MySQL regards TRUE and FALSE as no different from the ordinary integers 1 and 0). PostgreSQL additionally implements a IS UNKNOWN predicate, which can be used to test whether a three-value logical outcome is Unknown, although this is merely syntactic sugar.

BOOLEAN data type

The ISO SQL:1999 standard introduced the BOOLEAN data type to SQL, however it's still just an optional, non-core feature, coded T031.

When restricted by a NOT NULL constraint, the SQL BOOLEAN works like the Boolean type from other languages. Unrestricted however, the BOOLEAN datatype, despite its name, can hold the truth values TRUE, FALSE, and UNKNOWN, all of which are defined as boolean literals according to the standard. The standard also asserts that NULL and UNKNOWN "may be used interchangeably to mean exactly the same thing".[5]

The Boolean type has been subject of criticism, particularly because of the mandated behavior of the UNKNOWN literal, which is never equal to itself because of the identification with NULL.

As discussed above, in the PostgreSQL implementation of SQL, the null value is used to represent all UNKNOWN results, including the UNKNOWN BOOLEAN. PostgreSQL does not implement the UNKNOWN literal (although it does implement the IS UNKNOWN operator, which is an orthogonal feature.) Most other major vendors do not support the Boolean type (as defined in T031) as of 2012.[6] The procedural part of Oracle's PL/SQL supports BOOLEAN however variables; these can also be assigned NULL and the value is considered the same as UNKNOWN.

Controversy

Common mistakes

Misunderstanding of how Null works is the cause of a great number of errors in SQL code, both in ISO standard SQL statements and in the specific SQL dialects supported by real-world database management systems. These mistakes are usually the result of confusion between Null and either 0 (zero) or an empty string (a string value with a length of zero, represented in SQL as `''`). Null is defined by the ISO SQL standard as different from both an empty string and the numerical value 0, however. While Null indicates the absence of any value, the empty string and numerical zero both represent actual values.

A classic rookie error is attempting to use the equality operator to find NULL values. Most SQL implementations will execute the following query as syntactically correct (therefore give no error message) but it never returns any rows, regardless of whether NULL values do exist in the table.

```
SELECT *
FROM sometable
WHERE num = NULL;   -- Should be "WHERE num IS NULL"
```

In a related, but more subtle example, a `WHERE` clause or conditional statement might compare a column's value with a constant. It is often incorrectly assumed that a missing value would be "less than" or "not equal to" a constant if that field contains Null, but, in fact, such expressions return Unknown. An example is below:

```
SELECT *
FROM sometable
WHERE num <> 1;   -- Rows where num is NULL will not be returned,
                  -- contrary to many users' expectations.
```

Similarly, Null values are often confused with empty strings. Consider the `LENGTH` function, which returns the number of characters in a string. When a Null is passed into this function, the function returns Null. This can lead to unexpected results, if users are not well versed in 3-value logic. An example is below:

```
SELECT *
FROM sometable
WHERE LENGTH(string) < 20; -- Rows where string is NULL will not be returned.
```

This is complicated by the fact that in some database interface programs (or even database implementations like Oracle's), NULL is reported as an empty string, and empty strings may be incorrectly stored as NULL.

Criticisms

The ISO SQL implementation of Null is the subject of criticism, debate and calls for change. In *The Relational Model for Database Management: Version 2*, Codd suggested that the SQL implementation of Null was flawed and should be replaced by two distinct Null-type markers. The markers he proposed were to stand for *"Missing but Applicable"* and *"Missing but Inapplicable"*, known as *A-values* and *I-values*, respectively. Codd's recommendation, if accepted, would have required the implementation of a four-valued logic in SQL. Others have suggested adding additional Null-type markers to Codd's recommendation to indicate even more reasons that a data value might be "Missing", increasing the complexity of SQL's logic system. At various times, proposals have also been put forth to implement multiple user-defined Null markers in SQL. Because of the complexity of the Null-handling and logic systems required to support multiple Null markers, none of these proposals have gained widespread acceptance.

Chris Date and Hugh Darwen, authors of *The Third Manifesto*, have suggested that the SQL Null implementation is inherently flawed and should be eliminated altogether, pointing to inconsistencies and flaws in the implementation of SQL Null-handling (particularly in aggregate functions) as proof that the entire concept of Null is flawed and should be removed from the relational model. Others, like author Fabian Pascal, have stated a belief that "how the function calculation should treat missing values is not governed by the relational model."[*citation needed*]

Closed world assumption

Another point of conflict concerning Nulls is that they violate the closed world assumption model of relational databases by introducing an open world assumption into it. The closed world assumption, as it pertains to databases, states that "Everything stated by the database, either explicitly or implicitly, is true; everything else is false." This view assumes that the knowledge of the world stored within a database is complete. Nulls, however, operate under the open world assumption, in which some items stored in the database are considered unknown, making the database's stored knowledge of the world incomplete.

References

[1] Ron van der Meyden, " Logical approaches to incomplete information: a survey (http://books.google.com/books?id=gF0b85IuqQwC&pg=PA344)" in Chomicki, Jan; Saake, Gunter (Eds.) *Logics for Databases and Information Systems*, Kluwer Academic Publishers ISBN 978-0-7923-8129-7, p. 344; PS preprint (http://www.cse.unsw.edu.au/~meyden/research/indef-review.ps) (note: page numbering differs in preprint from the published version)

[2] C.J. Date (2004), *An introduction to database systems*, 8th ed., Pearson Education, p. 594

[3] C. J. Date, *Relational database writings, 1991-1994*, Addison-Wesley, 1995, p. 371

[4] C.J. Date (2004), *An introduction to database systems*, 8th ed., Pearson Education, p. 584

[5] ISO/IEC 9075-2:2011 •4.5

[6] Troels Arvin, Survey of BOOLEAN data type implementation (http://troels.arvin.dk/db/rdbms/#data_types-boolean)

Further reading

- E. F. Codd. Understanding relations (installment #7). FDT Bulletin of ACM-SIGMOD, 7(3-4):23€28, 1975.
- Codd, E. F. (1979). "Extending the database relational model to capture more meaning". *ACM Transactions on Database Systems* **4** (4): 397. doi: 10.1145/320107.320109 (http://dx.doi.org/10.1145/320107.320109). Especially •2.3.
- Date, C.J. (2000). *The Database Relational Model: A Retrospective Review and Analysis: A Historical Account and Assessment of E. F. Codd's Contribution to the Field of Database Technology*. Addison Wesley Longman. ISBN‚0-201-61294-1.
- Klein, Hans-Joachim. " How to modify SQL queries in order to guarantee sure answers (http://www.acm.org/sigmod/record/issues/9409/sql.ps)". ACM SIGMOD Record 23.3 (1994): 14-20.
- Claude Rubinson, Nulls, Three-Valued Logic, and Ambiguity in SQL: Critiquing Date,s Critique (http://www.u.arizona.edu/~rubinson/scrawl/Rubinson.2007.Nulls_Three-Valued_Logic_and_Ambiguity_in_SQL.pdf),

SIGMOD Record, December 2007 (Vol. 36, No. 4)
- € John Grant, Null Values in SQL (http://www09.sigmod.org/sigmod/record/issues/0809/p23.grant.pdf). SIGMOD Record, September 2008 (Vol. 37, No. 3)
- € Waraporn, Narongrit, and Kriengkrai Porkaew. " Null semantics for subqueries and atomic predicates (http://www.iaeng.org/IJCS/issues_v35/issue_3/IJCS_35_3_08.pdf)". IAENG International Journal of Computer Science 35.3 (2008): 305-313.
- € Bernhard Thalheim, Klaus-Dieter Schewe, "NULL †Value, Algebras and Logics" in Anneli Heimb‘rger, Yasushi Kiyoki, Takehiro Tokuda, Hannu Jaakkola, Naofumi Yoshida (eds.) Information Modelling and Knowledge Bases XXII, Frontiers in Artificial Intelligence and Applications, Volume 225, 2011, IOS Press, ISBN 978-1-60750-689-8, pp.ƒ354€367 doi: 10.3233/978-1-60750-690-4-354 (http://dx.doi.org/10.3233/978-1-60750-690-4-354)
- € Enrico Franconi and Sergio Tessaris, On the Logic of SQL Nulls (http://ceur-ws.org/Vol-866/paper8.pdf), Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management, Ouro Preto, Brazil, June 27€30, 2012. pp.ƒ114€128

External links

- € Oracle NULLs (http://www.psoug.org/reference/null.html)
- € The Third Manifesto (http://www.thethirdmanifesto.com/)
- € Implications of NULLs in sequencing of data (http://www.sqlexpert.co.uk/2006/05/treatment-of-nulls-by-oracle-sql.html)
- € Java bug report about jdbc not distinguishing null and empty string, which Sun closed as "not a bug" (http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4032732)
- € TheIntegrationEngineer (http://www.theintegrationengineer.com/the-nature-of-null/) explains how NULL works and the logic behind it.

Candidate key

In the relational model of databases, a **candidate key** of a relation is a minimal superkey for that relation; that is, a set of attributes such that

1. the relation does not have two distinct tuples (i.e. rows or records in common database language) with the same values for these attributes (which means that the set of attributes is a superkey)
2. there is no proper subset of these attributes for which (1) holds (which means that the set is minimal).

The constituent attributes are called **prime attributes**. Conversely, an attribute that does not occur in ANY candidate key is called a **non-prime attribute**.

Since a relation contains no duplicate tuples, the set of all its attributes is a superkey if NULL values are not used. It follows that every relation will have at least one candidate key.

The candidate keys of a relation tell us all the possible ways we can identify its tuples. As such they are an important concept for the design of database schema.

Example

The definition of candidate keys can be illustrated with the following (abstract) example. Consider a relation variable (relvar) *R* with attributes (*A*, *B*, *C*, *D*) that has only the following two legal values *r1* and *r2*:

r1

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d1
a2	b1	c2	d1

r2

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d1
a1	b1	c2	d2

Here *r2* differs from *r1* only in the **A** and **D** values of the last tuple.

For *r1* the following sets have the uniqueness property, i.e., there are no two distinct tuples in the instance with the same values for the attributes in the set:

{A,B}, {A,C}, {B,C}, {A,B,C}, {A,B,D}, {A,C,D}, {B,C,D}, {A,B,C,D}

For *r2* the uniqueness property holds for the following sets;

{B,C}, {B,D}, {C,D}, {A,B,C}, {A,B,D}, {A,C,D}, {B,C,D}, {A,B,C,D}

Since superkeys of a relvar are those sets of attributes that have the uniqueness property for *all* legal values of that relvar and because we assume that *r1* and *r2* are all the legal values that *R* can take, we can determine the set of superkeys of *R* by taking the intersection of the two lists:

{B,C}, {A,B,C}, {A,B,D}, {A,C,D}, {B,C,D}, {A,B,C,D}

Finally we need to select those sets for which there is no proper subset in the list, which are in this case:

{B,C}, {A,B,D}, {A,C,D}

These are indeed the candidate keys of relvar *R*.

We have to consider *all* the relations that might be assigned to a relvar to determine whether a certain set of attributes is a candidate key. For example, if we had considered only *r1* then we would have concluded that {A,B} is a candidate key, which is incorrect. However, we *might* be able to conclude from such a relation that a certain set is *not* a candidate key, because that set does not have the uniqueness property (example {A,D} for *r1*). Note that the existence of a proper subset of a set that has the uniqueness property *cannot* in general be used as evidence that the superset is not a candidate key. In particular, note that in the case of an empty relation, every subset of the heading has the uniqueness property, including the empty set.

Determining candidate keys

The set of all candidate keys can be computed e.g. from the set of functional dependencies. To this end we need to define the attribute closure α^+ for an attribute set α. The set α^+ contains all attributes that are functionally implied by α.

It is quite simple to find a single candidate key. We start with a set α of attributes and try to remove successively each attribute. If after removing an attribute the attribute closure stays the same, then this attribute is not necessary and we can remove it permanently. We call the result $\text{minimize}(\alpha)$. If α is the set of all attributes, then $\text{minimize}(\alpha)$ is a candidate key.

Actually we can detect every candidate key with this procedure by simply trying every possible order of removing attributes. However there are much more permutations of attributes ($n!$) than subsets (2^n). That is, many attribute orders will lead to the same candidate key.

There is a fundamental difficulty for efficient algorithms for candidate key computation: Certain sets of functional dependencies lead to exponentially many candidate keys. Consider the $2 \cdot n$ functional dependencies $\{A_i \to B_i : i \in \{1, \ldots, n\}\} \cup \{B_i \to A_i : i \in \{1, \ldots, n\}\}$ which yields 2^n candidate keys: $\{A_1, B_1\} \times \ldots \times \{A_n, B_n\}$. That is, the best we can expect is an algorithm that is efficient with respect to the number of candidate keys.

The following algorithm actually runs in polynomial time in the number of candidate keys and functional dependencies:

```
K[0] := minimize(A);  /* A is the set of all attribute */
n := 1; /* Number of Keys known so far */
i := 0; /* Currently processed key */
while i < n do
  foreach • … , ˆ F do
    S := • •(K[i] „ ,);
    found := false;
    for j := 0 to n-1 do
      if K[j] •S then found := true;
    if not found then
      K[n] := minimize(S);
      n := n + 1;
```

The idea behind the algorithm is that given a candidate key K_i and a functional dependency $\alpha \to \beta$, the reverse application of the functional dependency yields the set $\alpha \cup (K_i \setminus \beta)$, which is a key, too. It may however be covered by other already known candidate keys. (The algorithm checks this case using the 'found' variable.) If not, then minimizing the new key yields a new candidate key. The key insight is (pun not intended) that all candidate keys can be created this way.

References

€ Date, Christopher (2003). "5: Integrity". *An Introduction to Database Systems*. Addison-Wesley. pp.ƒ268€276. ISBNƒ978-0-321-18956-1.

External links

€ Relational Database Management Systems - Database Design - Terms of Reference - Keys (http://rdbms.opengrass.net/2_Database Design/2.1_TermsOfReference/2.1.2_Keys.html): An overview of the different types of keys in an RDBMS (Relational Database Management System).

Foreign key

In the context of relational databases, a **foreign key** is a field (or collection of fields) in one table that uniquely identifies a row of another table. In other words, a foreign key is a column or a combination of columns that is used to establish and enforce a link between the data in two tables.

For example, consider a database with two tables, a CUSTOMER table that includes all customer data and an ORDER table that includes all customer orders. Suppose that the business requires that each order must refer to a single customer. To reflect this in the database, the primary key (e.g., CUSTOMERID) in the CUSTOMER table is added to the ORDER table, where it is called a foreign key. Since CUSTOMERID in the ORDER table uniquely identifies a row of the CUSTOMER table, it says which customer placed the order.

The table containing the foreign key is called the referencing or child table and the table containing the candidate key is called the referenced or parent table. Since the purpose of the foreign key in the referencing table is to identify a row of the referenced table, the value of the foreign key must be equal to the candidate key's value in some row of the primary table or else have no value, i.e., the NULL value. This rule is called a referential integrity constraint between the two tables. Because violations of referential integrity constraints can be the source of many database problems, most database management systems enforce referential integrity constraints, providing mechanisms to ensure that every non-null foreign key corresponds to a row of the referenced (or parent) table.

Foreign keys play an essential role in database design. One important part of database design is making sure that relationships between real-world entities are reflected in the database by references, using foreign keys to refer from one table to another. Another important part of database design is database normalization, in which tables are broken apart and foreign keys make it possible for them to be reconstructed.

Multiple rows in the referencing (or child) table may refer to the same row in the referenced (or parent) table. For this reason, the relationship between the two tables is called a one to many relationship between the referenced table and the referencing table. The child and parent table may be the same table, i.e. the foreign key refers back to the same table. Such a foreign key is known in SQL:2003 as a **self-referencing** or **recursive** foreign key.

A table may have multiple foreign keys, and each foreign key can have a different parent table. Each foreign key is enforced independently by the database system. Therefore, cascading relationships between tables can be established using foreign keys.

Defining foreign keys

Foreign keys are defined in the ISO SQL Standard, through a FOREIGN KEY constraint. The syntax to add such a constraint to an existing table is defined in SQL:2003 as shown below. Omitting the column list in the REFERENCES clause implies that the foreign key shall reference the primary key of the referenced table.

```
ALTER TABLE <table identifier>
   ADD [ CONSTRAINT <constraint identifier> ]
      FOREIGN KEY ( <column expression> {, <column expression>}... )
      REFERENCES <table identifier> [ ( <column expression> {, <column expression>}... ) ]
      [ ON UPDATE <referential action> ]
      [ ON DELETE <referential action> ]
```

Likewise, foreign keys can be defined as part of the CREATE TABLE SQL statement.

```
CREATE TABLE table_name (
   id    INTEGER  PRIMARY KEY,
   col2  CHARACTER VARYING(20),
   col3  INTEGER,
   ...
   FOREIGN KEY(col3)
      REFERENCES other_table(key_col) ON DELETE CASCADE,
   ... )
```

If the foreign key is a single column only, the column can be marked as such using the following syntax:

```
CREATE TABLE table_name (
   id    INTEGER  PRIMARY KEY,
   col2  CHARACTER VARYING(20),
   col3  INTEGER REFERENCES other_table(column_name),
   ... )
```

Foreign keys can be defined with a stored procedure statement.Wikipedia:Please clarify

```
sp_foreignkey tabname, pktabname, col1 [, col2] ...  [, col8]
```

- **tabname**: the name of the table or view that contains the foreign key to be defined.
- **pktabname**: the name of the table or view that has the primary key to which the foreign key applies. The primary key must already be defined.
- **col1**: the name of the first column that makes up the foreign key. The foreign key must have at least one column and can have a maximum of eight columns.

Referential actions

Because the database management system enforces referential constraints, it must ensure data integrity if rows in a referenced table are to be deleted (or updated). If dependent rows in referencing tables still exist, those references have to be considered. SQL:2003 specifies 5 different **referential actions** that shall take place in such occurrences:

- CASCADE
- RESTRICT
- NO ACTION
- SET NULL
- SET DEFAULT

CASCADE

Whenever rows in the master (referenced) table are deleted (resp. updated), the respective rows of the child (referencing) table with a matching foreign key column will get deleted (resp. updated) as well. This is called a cascade delete (resp. update).

RESTRICT

A value cannot be updated or deleted when a row exists in a referencing or child table that references the value in the referenced table.

Similarly, a row cannot be deleted as long as there is a reference to it from a referencing or child table.

NO ACTION

NO ACTION and RESTRICT are very much alike. The main difference between NO ACTION and RESTRICT is that with NO ACTION the referential integrity check is done after trying to alter the table. RESTRICT does the check before trying to execute the UPDATE or DELETE statement. Both referential actions act the same if the referential integrity check fails: the UPDATE or DELETE statement will result in an error.

In other words, when an UPDATE or DELETE statement is executed on the referenced table using the referential action NO ACTION, the DBMS verifies at the end of the statement execution that none of the referential relationships are violated. This is different from RESTRICT, which assumes at the outset that the operation will violate the constraint. Using NO ACTION, the triggers or the semantics of the statement itself may yield an end state in which no foreign key relationships are violated by the time the constraint is finally checked, thus allowing the statement to complete successfully.

SET DEFAULT , SET NULL

In general, the action taken by the DBMS for SET NULL or SET DEFAULT is the same for both ON DELETE or ON UPDATE: The value of the affected referencing attributes is changed to NULL for SET NULL, and to the specified default value for SET DEFAULT.

Triggers

Referential actions are generally implemented as implied triggers (i.e. triggers with system-generated names, often hidden.) As such, they are subject to the same limitations as user-defined triggers, and their order of execution relative to other triggers may need to be considered; in some cases it may become necessary to replace the referential action with its equivalent user-defined trigger to ensure proper execution order, or to work around mutating-table limitations.

Another important limitation appears with transaction isolation: your changes to a row may not be able to fully cascade because the row is referenced by data your transaction cannot "see", and therefore cannot cascade onto. An

example: while your transaction is attempting to renumber a customer account, a simultaneous transaction is attempting to create a new invoice for that same customer; while a CASCADE rule may fix all the invoice rows your transaction can see to keep them consistent with the renumbered customer row, it won't reach into another transaction to fix the data there; because the database cannot guarantee consistent data when the two transactions commit, one of them will be forced to roll back (often on a first-come-first-served basis.)

Example

As a first example to illustrate foreign keys, suppose an accounts database has a table with invoices and each invoice is associated with a particular supplier. Supplier details (such as name and address) are kept in a separate table; each supplier is given a 'supplier number' to identify it. Each invoice record has an attribute containing the supplier number for that invoice. Then, the 'supplier number' is the primary key in the Supplier table. The foreign key in the Invoices table points to that primary key. The relational schema is the following. Primary keys are marked in bold, and foreign keys are marked in italics.

```
Supplier ( SupplierNumber, Name, Address, Type )
Invoices ( InvoiceNumber, SupplierNumber, Text )
```

The corresponding Data Definition Language statement is as follows.

```
CREATE TABLE Supplier (
   SupplierNumber   INTEGER NOT NULL,
   Name             VARCHAR(20) NOT NULL,
   Address          VARCHAR(50) NOT NULL,
   Type             VARCHAR(10),
   CONSTRAINT supplier_pk PRIMARY KEY(SupplierNumber),
   CONSTRAINT number_value CHECK (SupplierNumber > 0) )

CREATE TABLE Invoices (
   InvoiceNumber    INTEGER NOT NULL,
   SupplierNumber   INTEGER NOT NULL,
   Text             VARCHAR(4096),
   CONSTRAINT invoice_pk PRIMARY KEY(InvoiceNumber),
   CONSTRAINT inumber_value CHECK (InvoiceNumber > 0),
   CONSTRAINT supplier_fk FOREIGN KEY(SupplierNumber)
      REFERENCES Supplier(SupplierNumber)
      ON UPDATE CASCADE ON DELETE RESTRICT )
```

References

External links

- SQL-99 Foreign Keys (https://kb.askmonty.org/en/constraint_type-foreign-key-constraint/)
- PostgreSQL Foreign Keys (http://www.postgresql.org/docs/9.2/static/tutorial-fk.html)
- MySQL Foreign Keys (http://dev.mysql.com/doc/refman/5.1/en/create-table-foreign-keys.html)
- FirebirdSQL Foreign Keys (http://www.firebirdsql.org/manual/nullguide-keys.html#nullguide-keys-fk)
- SQLite support for Foreign Keys (http://www.sqlite.org/foreignkeys.html)

Unique key

In an entity relationship diagram of a data model, one or more **unique keys** may be declared for each data entity. Each unique key is composed from one or more data attributes of that data entity. The set of unique keys declared for a data entity is often referred to as the candidate keys for that data entity. From the set of candidate keys, a single unique key is selected and declared the primary key for that data entity. In an entity relationship diagram, each entity relationship uses a unique key, most often the primary key, of one data entity and copies the unique key data attributes to another data entity to which it relates. This inheritance of the unique key data attributes is referred to as a foreign key and is used to provide data access paths between data entities. Once the data model is instantiated into a database, each data entity usually becomes a database table, unique keys become unique indexes associated with their assigned database tables, and entity relationships become foreign key constraints. In integrated data models,[1] commonality relationships[2] do not become foreign key constraints since commonality relationships are a peer-to-peer type of relationship.

In a relational database, a "Primary Key" is a key that uniquely defines the characteristics of each row (also known as record or tuple). The **primary key** has to consist of characteristics that cannot be duplicated by any other row. The primary key may consist of a single attribute or a multiple attributes in combination. For example, a birthday could be shared by many people and so would not be a prime candidate for the Primary Key, but a social security number or Driver's License number would be ideal since it correlates to one single data value. Another unique characteristic of a Primary Key as it pertains to a relational database, is that a Primary Key must also serve as a Foreign Key on a related table[*citation needed*]. For example:

```
Author Table Schema:

AuthorTable(AUTHOR_ID,AuthorName,CountryBorn,YearBorn)

Book Table Schema:

Book Table(ISBN,Author_ID,Title,Publisher,Price)
```

Here we can see that AUTHOR_ID serves as the Primary Key in AuthorTable but also serves as the Foreign Key on the BookTable. The Foreign Key serves as the link and therefore the connection between the two "related" tables in this sample database.

In a relational database, a **unique key** index can uniquely identify each row of data values in a database table. A unique key index comprises a single column or a set of columns in a single database table. No two distinct rows or data records in a database table can have the same data value (or combination of data values) in those unique key index columns if NULL values are not used. Depending on its design, a database table may have many unique key indexes but at most one primary key index.

A unique key constraint does not imply the NOT NULL constraint in practice. Because NULL is not an actual value (it represents the lack of a value), when two rows are compared, and both rows have NULL in a column, the column values are not considered to be equal. Thus, in order for a unique key to uniquely identify each row in a table, NULL values must not be used. According to the SQL[3] standard and Relational Model theory, a unique key (unique constraint) should accept NULL in several rows/tuples ƒ however not all RDBMS implement this feature correctly.[4][5]

A unique key should uniquely identify all *possible* rows that exist in a table and not only the currently existing rows [*citation needed*]. Examples of unique keys are Social Security numbers (associated with a specific person[6]) or ISBNs (associated with a specific book). Telephone books and dictionaries cannot use names, words, or Dewey Decimal system numbers as candidate keys because they do not uniquely identify telephone numbers or words.

A table can have at most one **primary key**, but more than one unique key. A primary key is a combination of columns which uniquely specify a row. It is a special case of unique keys. One difference is that primary keys have an implicit NOT NULL constraint while unique keys do not. Thus, the values in unique key columns may or may not be NULL, and in fact such a column may contain at most one NULL fields.[7] Another difference is that primary keys must be defined using another syntax.

The relational model, as expressed through relational calculus and relational algebra, does not distinguish between primary keys and other kinds of keys. Primary keys were added to the SQL standard mainly as a convenience to the application programmer.[*citation needed*]

Unique keys as well as primary keys can be referenced by foreign keys.

Defining primary keys

Primary keys are defined in the ANSI SQL Standard, through the PRIMARY KEY constraint. The syntax to add such a constraint to an existing table is defined in SQL:2003 like this:

```
ALTER TABLE <table identifier>
    ADD [ CONSTRAINT <constraint identifier> ]
    PRIMARY KEY ( <column expression> {, <column expression>}... )
```

The primary key can also be specified directly during table creation. In the SQL Standard, primary keys may consist of one or multiple columns. Each column participating in the primary key is implicitly defined as NOT NULL. Note that some DBMS require explicitly marking primary-key columns as NOT NULL.[*citation needed*]

```
CREATE TABLE table_name (

   ...
)
```

If the primary key consists only of a single column, the column can be marked as such using the following syntax:

```
CREATE TABLE table_name (
   id_col  INT  PRIMARY KEY,
   col2    CHARACTER VARYING(20),
   ...
)
```

Differences between Primary Key and Unique Key:

Primary Key

1. A primary key *cannot allow* null values. (You cannot define a primary key on columns that allow nulls.)
2. Each table can have *at most one* primary key.
3. On some RDBMS a primary key automatically generates a *clustered* table index by default.

Unique Key

1. A unique key *can allow* null values. (You can define a unique key on columns that allow nulls.)
2. Each table can have *multiple* unique keys.
3. On some RDBMS a unique key automatically generates a *non-clustered* table index by default.

Defining unique keys

The definition of unique keys is syntactically very similar to primary keys.

```
ALTER TABLE <table identifier>
    ADD [ CONSTRAINT <constraint identifier> ]
    UNIQUE ( <column expression> {, <column expression>}... )
```

Likewise, unique keys can be defined as part of the CREATE TABLE SQL statement.

```
CREATE TABLE table_name (
   id_col   INT,
   col2     CHARACTER VARYING(20),
   key_col  SMALLINT,
   ...
   CONSTRAINT key_unique UNIQUE(key_col),
   ...
)
```

```
CREATE TABLE table_name (
   id_col  INT  PRIMARY KEY,
   col2    CHARACTER VARYING(20),
   ...
   key_col  SMALLINT UNIQUE,
   ...
)
```

Surrogate keys

In some design situations the natural key that uniquely identifies a tuple in a relation is difficult to use for software development. For example, it may involve multiple columns or large text fields. A surrogate key can be used as the primary key. In other situations there may be more than one candidate key for a relation, and no candidate key is obviously preferred. A surrogate key may be used as the primary key to avoid giving one candidate key artificial primacy over the others.

Since primary keys exist primarily as a convenience to the programmer, surrogate primary keys are often used *ƒ* in many cases exclusively *ƒ* in database application design.

Due to the popularity of surrogate primary keys, many developers and in some cases even theoreticians have come to regard surrogate primary keys as an inalienable part of the relational data model. This is largely due to a migration of principles from the Object-Oriented Programming model to the relational model, creating the hybrid object-relational model. In the ORM, these additional restrictions are placed on primary keys:

- Primary keys should be immutable, that is, not changed until the record is destroyed.
- Primary keys should be anonymous integer or numeric identifiers.

However, neither of these restrictions is part of the relational model or any SQL standard. Due diligence should be applied when deciding on the immutability of primary key values during database and application design. Some database systems even imply that values in primary key columns cannot be changed using the UPDATE SQL statement[*citation needed*].

Alternate key

It is commonplace in SQL databases to declare a single **primary key**, the most important unique key. However, there could be further unique keys that could serve the same purpose. These should be marked as 'unique' keys. This is done to prevent incorrect data from entering a table (a duplicate entry is not valid in a unique column) and to make the database more complete and useful. These could be called alternate keys.[8]

References

[1] Data Model Integration | The Integration of Data Models (http://www.strins.com/data-model-integration.html)

[2] Commonality Relationships | Commonality Constraints (http://www.strins.com/commonality-relationships.html)

[3] Summary of ANSI/ISO/IEC SQL (http://www.xcdsql.org/Summary of SQL.html#chapter-Table constraints)

[4] Constraints - SQL Database Reference Material - Learn sql, read an sql manual, follow an sql tutorial, or learn how to structure an SQL query (http://www.sql.org/sql-database/postgresql/manual/ddl-constraints.html#AEN1832)

[5] Comparison of different SQL implementations (http://troels.arvin.dk/db/rdbms/#constraints-unique)

[6] **SSN uniqueness:** Rare SSN duplicates do exist in the field, a condition that led to problems with early commercial computer systems that relied on SSN uniqueness. Practitioners are taught that well-known duplications in SSN assignments occurred in the early days of the SSN system. This situation points out the complexity of designing systems that assume unique keys in real-world data.

[7] MySQL 5.5 Reference Manual :: 12.1.14. CREATE TABLE Syntax (http://dev.mysql.com/doc/refman/5.5/en/create-table.html) "For all engines, a UNIQUE index permits multiple NULL values for columns that can contain NULL."

[8] Alternate key - Oracle FAQ (http://www.orafaq.com/wiki/Alternate_key)

External links

- Relation Database terms of reference, Keys (http://rdbms.opengrass.net/2_Database Design/2. 1_TermsOfReference/2.1.2_Keys.html): An overview of the different types of keys in an RDBMS

Superkey

A **superkey** is defined in the relational model of database organization as a set of attributes of a relation variable for which it holds that in all relations assigned to that variable, there are no two distinct tuples (rows) that have the same values for the attributes in this set. Equivalently a superkey can also be defined as a set of attributes of a relation schema upon which all attributes of the schema are functionally dependent.

Note that the set of **all** attributes is a trivial superkey, because in relational algebra duplicate rows are not permitted.

Also note that if attribute set *K* is a superkey of relation *R*, then at all times it is the case that the projection of *R* over *K* has the same cardinality as *R* itself.

Informally, a superkey is a set of attributes within a table whose values can be used to uniquely identify a tuple. A candidate key is a minimal set of attributes necessary to identify a tuple, this is also called a minimal superkey. For example, given an employee schema, consisting of the attributes employeeID, name, job, and departmentID, we could use the employeeID in combination with any or all other attributes of this table to uniquely identify a tuple in the table. Examples of superkeys in this schema would be {employeeID, Name}, {employeeID, Name, job}, and {employeeID, Name, job, departmentID}. The last example is known as trivial superkey, because it uses all attributes of this table to identify the tuple.

In a real database we do not need values for all of those attributes to identify a tuple. We only need, per our example, the set {employeeID}. This is a **minimal superkey**ƒ€ that is, a minimal set of attributes that can be used to identify a single tuple. So, employeeID is a candidate key.

Example

English Monarchs

Monarch Name	Monarch Number	Royal House
Edward	II	Plantagenet
Edward	III	Plantagenet
Richard	III	Plantagenet
Henry	IV	Lancaster

First, list out all the (non-empty) sets of attributes:

- {Monarch Name}
- {Monarch Number}
- {Royal House}
- {Monarch Name, Monarch Number}
- {Monarch Name, Royal House}
- {Monarch Number, Royal House}
- {Monarch Name, Monarch Number, Royal House}

Second, eliminate all the sets which **do not** meet superkey's requirement. For example, {Monarch Name, Royal House} cannot be a superkey because for the same attribute values (Edward, Plantagenet), there are two distinct tuples:

- (Edward, **II**, Plantagenet)
- (Edward, **III**, Plantagenet)

Finally, after elimination, the remaining sets of attributes are the only possible superkeys in this example:

- {Monarch Name, Monarch Number} **(Candidate Key)**
- {Monarch Name, Monarch Number, Royal House}

In real situations, however, superkeys are normally not determined by this method, which is very tedious and time-consuming, but by analyzing functional dependencies (FD).

References

- Silberschatz, Abraham (2011). *Database System Concepts (6th ed.)*. McGraw-Hill. pp.⁄45€46. ISBN⁄978-0-07-352332-3.

External links

- Relation Database terms of reference, Keys (http://rdbms.opengrass.net/2_Database Design/2.1_TermsOfReference/2.1.2_Keys.html): An overview of the different types of keys in an RDBMS

Surrogate key

A **surrogate key** in a database is a unique identifier for either an *entity* in the modeled world or an *object* in the database. The surrogate key is *not* derived from application data.

Definition

There are at least two definitions of a surrogate:

Surrogate (1) € Hall, Owlett and Codd (1976)

> A surrogate represents an *entity* in the outside world. The surrogate is internally generated by the system but is nevertheless visible to the user or application.

Surrogate (2) € Wieringa and De Jonge (1991)

> A surrogate represents an *object* in the database itself. The surrogate is internally generated by the system and is invisible to the user or application.

The *Surrogate (1)* definition relates to a data model rather than a storage model and is used throughout this article. See Date (1998).

An important distinction between a surrogate and a primary key depends on whether the database is a current database or a temporal database. Since a *current database* stores only *currently* valid data, there is a one-to-one correspondence between a surrogate in the modeled world and the primary key of the database. In this case the surrogate may be used as a primary key, resulting in the term *surrogate key*. In a temporal database, however, there is a many-to-one relationship between primary keys and the surrogate. Since there may be several objects in the database corresponding to a single surrogate, we cannot use the surrogate as a primary key; another attribute is required, in addition to the surrogate, to uniquely identify each object.

Although Hall *et al.* (1976) say nothing about this, othersWikipedia:Citing sources have argued that a surrogate should have the following characteristics:

- the value is unique system-wide, hence never reused
- the value is system generated
- the value is not manipulable by the user or application
- the value contains no semantic meaning
- the value is not visible to the user or application
- the value is not composed of several values from different domains.

Surrogates in practice

In a current database, the surrogate key can be the primary key, generated by the database management system and *not* derived from any application data in the database. The only significance of the surrogate key is to act as the primary key. It is also possible that the surrogate key exists in addition to the database-generated UUID (for example, an HR number for each employee other than the UUID of each employee).

A surrogate key is frequently a sequential number (e.g. a Sybase or SQL Server "identity column", a PostgreSQL or Informix `serial`, an Oracle SEQUENCE or a column defined with AUTO_INCREMENT in MySQL) but doesn't have to be. Having the key independent of all other columns insulates the database relationships from changes in data values or database design (making the database more agile) and guarantees uniqueness.

In a temporal database, it is necessary to distinguish between the surrogate key and the primary key. Typically, every row would have both a primary key and a surrogate key. The primary key identifies the unique row in the database, the surrogate key identifies the unique entity in the modelled world; these two keys are not the same. For example, table *Staff* may contain two rows for "John Smith", one row when he was employed between 1990 and 1999, another

row when he was employed between 2001 and 2006. The surrogate key is identical (non-unique) in both rows however the primary key *will* be unique.

Some database designers use surrogate keys systematically regardless of the suitability of other candidate keys, while others will use a key already present in the data, if there is one.

A *surrogate key* may also be called a synthetic key, an entity identifier, a system-generated key, a database sequence number, a factless key, a technical key, or an arbitrary unique identifier.[*citation needed*] Some of these terms describe the way of *generating* new surrogate values rather than the *nature* of the surrogate concept.

Approaches to generating surrogates include:

- Universally Unique Identifiers (UUIDs)
- Globally Unique Identifiers (GUIDs)
- Object Identifiers (OIDs)
- Sybase or SQL Server identity column `IDENTITY` OR `IDENTITY(n,n)`
- Oracle SEQUENCE
- PostgreSQL or IBM Informix serial
- MySQL `AUTO_INCREMENT`
- AutoNumber data type in Microsoft Access
- `AS IDENTITY GENERATED BY DEFAULT` in IBM DB2
- Identity column (implemented in DDL) in Teradata

Advantages

Immutability

Surrogate keys do not change while the row exists. This has the following advantages:

- Applications cannot lose their reference to a row in the database (since the identifier never changes).
- The primary key data can always be modified, even with databases that do not support cascading updates across related foreign keys.

Requirement changes

Attributes that uniquely identify an entity might change, which might invalidate the suitability of the natural, compound keys. Consider the following example:

> An employee's network user name is chosen as a natural key. Upon merging with another company, new employees must be inserted. Some of the new network user names create conflicts because their user names were generated independently (when the companies were separate).

In these cases, generally a new attribute must be added to the natural key (for example, an *original_company* column). With a surrogate key, only the table that defines the surrogate key must be changed. With natural keys, all tables (and possibly other, related software) that use the natural key will have to change.

Some problem domains do not clearly identify a suitable natural key. Surrogate key avoids choosing a natural key that might be incorrect.

Performance

Surrogate keys tend to be a compact data type, such as a four-byte integer. This allows the database to query the single key column faster than it could multiple columns. Furthermore a non-redundant distribution of keys causes the resulting b-tree index to be completely balanced. Surrogate keys are also less expensive to join (fewer columns to compare) than compound keys.

Compatibility

While using several database application development systems, drivers, and object-relational mapping systems, such as Ruby on Rails or Hibernate, it is much easier to use an integer or GUID surrogate keys for every table instead of natural keys in order to support database-system-agnostic operations and object-to-row mapping.

Uniformity

When every table has a uniform surrogate key, some tasks can be easily automated by writing the code in a table-independent way.

Validation

It is possible to design key-values that follow a well-known pattern or structure which can be automatically verified. For instance, the keys that are intended to be used in some column of some table might be designed to "look differently from" those that are intended to be used in another column or table, thereby simplifying the detection of application errors in which the keys have been misplaced. However, this characteristic of the surrogate keys should never be used to drive any of the logic of the applications themselves, as this would violate the principles of Database normalization.

Disadvantages

Disassociation

The values of generated surrogate keys have no relationship to the real-world *meaning* of the data held in a row. When inspecting a row holding a foreign key reference to another table using a surrogate key, the meaning of the surrogate key's row cannot be discerned from the key itself. Every foreign key must be joined to see the related data item. This can also make auditing more difficult,[*citation needed*] as incorrect data is not obvious.

Surrogate keys are unnatural for data that is exported and shared. A particular difficulty is that tables from two otherwise identical schemas (for example, a test schema and a development schema) can hold records that are equivalent in a business sense, but have different keys. This can be mitigated by not exporting surrogate keys, except as transient data (most obviously, in executing applications that have a "live" connection to the database).

Query optimization

Relational databases assume a unique index is applied to a table's primary key. The unique index serves two purposes: (i) to enforce entity integrity, since primary key data must be unique across rows and (ii) to quickly search for rows when queried. Since surrogate keys replace a table's identifying attributes *ƒ* the natural key *ƒ* and since the identifying attributes are likely to be those queried, then the query optimizer is forced to perform a full table scan when fulfilling likely queries. The remedy to the full table scan is to apply indexes on the identifying attributes, or sets of them. Where such sets are themselves a candidate key, the index can be a unique index.

These additional indexes, however, will take up disk space and slow down inserts and deletes.

Normalization

The presence of a surrogate key can result in the database administrator forgetting to establish, or accidentally removing, a secondary unique index on the natural key of the table. Without a unique index on the natural key, duplicate rows can appear and once present can be difficult to identify.

Business process modeling

Because surrogate keys are unnatural, flaws can appear when modeling the business requirements. Business requirements, relying on the natural key, then need to be translated to the surrogate key. A strategy is to draw a clear distinction between the logical model (in which surrogate keys do not appear) and the physical implementation of that model, to ensure that the logical model is correct and reasonably well normalised, and to ensure that the physical model is a correct implementation of the logical model.

Inadvertent disclosure

Proprietary information can be leaked if sequential key generators are used. By subtracting a previously generated sequential key from a recently generated sequential key, one could learn the number of rows inserted during that time period. This could expose, for example, the number of transactions or new accounts per period. There are a few ways to overcome this problem:

- Increase the sequential number by a random amount.
- Generate a completely random primary key. However, to prevent duplication which would cause an insert rejection, a randomly generated primary key must either be queried (to check that it is not already in use), or the key must contain enough entropy that one can be confident that collisions will not happen.

Inadvertent assumptions

One might incorrectly infer from sequentially generated surrogate keys that events with a higher primary key value occurred after events with a lower primary key value. The sequential primary key implies nothing of the kind. It is possible for inserts to fail and leave gaps, and for those gaps to be filled at some later time. A sequential key value is not a reliable indicator of chronology. If chronology is important, rely not upon the sequential key but upon a timestamp. A random key would prevent a person from making the assumption that the key has some bearing to real-world chronology only if the person making the assumption is aware that the key is indeed random and has no bearing upon chronology. A randomly generated primary key must be queried before assigned to prevent duplication and cause an insert rejection.[*citation needed*]

References

This article is based on material taken from the Free On-line Dictionary of Computing prior to 1 November 2008 and incorporated under the "relicensing" terms of the GFDL, version 1.3 or later.

- Nijssen, G.M. (1976). *Modelling in Data Base Management Systems*. North-Holland Pub. Co. ISBN 0-7204-0459-2.
- Engles, R.W.: (1972), *A Tutorial on Data-Base Organization*, Annual Review in Automatic Programming, Vol.7, Part 1, Pergamon Press, Oxford, pp. 1–64.
- Langefors, B (1968). *Elementary Files and Elementary File Records*, Proceedings of File 68, an IFIP/IAG International Seminar on File Organisation, Amsterdam, November, pp. 89–96.
- Wieringa, R.; de Jonge, W. (1991). *The identification of objects and roles: Object identifiers revisited*. CiteSeerX: 10.1.1.16.3195 [1].
- Date, C. J. (1998). "Chapters 11 and 12". *Relational Database Writings 1994–1997*. ASIN 0201398141 [2].
- Carter, Breck. "Intelligent Versus Surrogate Keys" [3]. Retrieved 2006-12-03.

€ Richardson, Lee. "Create Data Disaster: Avoid Unique Indexes € (Mistake 3 of 10)" [4]. Retrieved 2008-01-19.

€ Berkus, Josh. "Database Soup: Primary Keyvil, Part I" [5]. Retrieved 2006-12-03.

References

[1] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.3195

[2] http://www.amazon.co.uk/dp/0201398141

[3] http://www.bcarter.com/intsurr1.htm

[4] http://www.nearinfinity.com/blogs/page/lrichard?entry=create_data_disaster_avoid_unique

[5] http://blogs.ittoolbox.com/database/soup/archives/primary-keyvil-part-i-7327

Armstrong's axioms

Armstrong's axioms are a set of axioms (or, more precisely, inference rules) used for infer all the functional dependencies on a relational database. They were developed by William W. Armstrong on his 1974 paper.[1] The axioms are sound in generating only functional dependencies in the closure of a set of functional dependencies (denoted as F^+) when applied to that set (denoted as F). They are also complete in that repeated application of these rules will generate all functional dependencies in the closure F^+.

More formally, let < R (U), F > denote a relational scheme over the set of attributes U with a set of functional dependencies F. We say that a functional dependency f is logically implied by F,and denote it with $F \models f$ if and only if for every instance r of R that satisfies the functional dependencies in F, r also satisfies f. We denote by F^+ the set of all functional dependencies that are logically implied by F.

Furthermore, with respect to a set of inference rules A , we say that a functional dependency f is derivable from the functional dependencies in F by the set of inference rules A , and we denote it by $F \vdash_A f$ if and only if f is obtainable by means of repeatedly applying the inference rules in A to functional dependencies in F. We denote by F_A^* the set of all functional dependencies that are derivable from F by inference rules in A .

Then, a set of inference rules A is sound if and only if the following holds:

$F_A^* \subseteq F^+$

that is to say, we cannot derive by means of A functional dependencies that are not logically implied by F. The set of inference rules A is said to be complete if the following holds:

$F^+ \subseteq F_A^*$

more simply put, we are able to derive by A all the functional dependencies that are logically implied by F.

Axioms

Let R (U) be a relation scheme over the set of attributes U . Henceforth we will denote by letters X , Y, Z any subset of U and, for short, the union of two sets of attributes X and Y by XY instead of the usual $X \cup Y$; this notation is rather standard in database theory when dealing with sets of attributes.

Axiom of Reflexivity

If $Y \subseteq X$, then $X \rightarrow Y$

Axiom of augmentation

If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z If $X \rightarrow Y$, then $XC \rightarrow YC$ for any C

Axiom of transitivity

If $X \to Y$ and $Y \to Z$, then $X \to Z$

Additional rules

These rules can be derived from above axioms.

Union

If $X \to Y$ and $X \to Z$ then $X \to YZ$

Decomposition

If $X \to YZ$ then $X \to Y$ and $X \to Z$

Pseudo transitivity

If $A \to B$ and $BC \to D$ then $AC \to D$

Armstrong relation

Given a set of functional dependencies F, the **Armstrong relation** is a relation which satisfies all the functional dependencies in the closure F^+ and only those dependencies. Unfortunately, the minimum-size Armstrong relation for a given set of dependencies can have a size which is an exponential function of the number of attributes in the dependencies considered.

External links

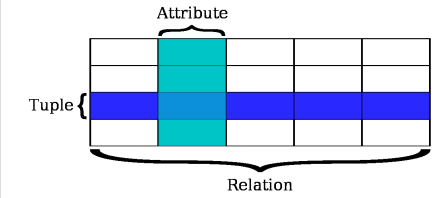
- UMBC CMSC 461 Spring '99 [2]
- CS345 Lecture Notes from Stanford University [3]

References

[1] William Ward Armstrong: *Dependency Structures of Data Base Relationships*, page 580-583. IFIP Congress, 1974.
[2] http://www.cs.umbc.edu/courses/461/current/burt/lectures/lec14/
[3] http://www-db.stanford.edu/~ullman/cs345notes/slides01-1.ps

Objects

Relation (database)



Relation, tuple, and attribute represented as table, row, and column.

In relational database theory, a **relation**, as originally defined by E.F. Codd, is a set of tuples $(d_1, d_2, ..., d_n)$, where each element d_j is a member of D_j, a data domain. Codd's original definition notwithstanding, and contrary to the usual definition in mathematics, there is no ordering to the elements of the tuples of a relation. Instead, each element is termed an **attribute value**. An **attribute** is a name paired with a domain (nowadays more commonly referred to as **type** or **data type**). An **attribute value** is an attribute name paired with an element of that attribute's domain, and a tuple is a *set* of attribute values in which no two distinct elements have the same name. Thus, in some accounts, a tuple is described as a function, mapping names to values.

A set of attributes in which no two distinct elements have the same name is called a **heading**. A set of tuples having the same heading is called a **body**. A relation is thus a heading paired with a body, the heading of the relation being also the heading of each tuple in its body. The number of attributes constituting a heading is called the **degree**, which term also applies to tuples and relations. The term *n*-tuple refers to a tuple of degree *n* (*n*>=0).

E. F. Codd used the term relation in its mathematical sense of a finitary relation, a set of tuples on some set of *n* sets $S_1, S_2, ... , S_n$. Thus, an *n*-ary relation is interpreted, under the Closed World Assumption, as the extension of some *n*-adic predicate: all and only those *n*-tuples whose values, substituted for corresponding free variables in the predicate, yield propositions that hold true, appear in the relation.

The term relation schema refers to a heading paired with a set of constraints defined in terms of that heading. A relation can thus be seen as an instantiation of a relation schema if it has the heading of that schema and it satisfies the applicable constraints.

Sometimes a relation schema is taken to include a name. A relational database definition (database schema, sometimes referred to as a relation*al* schema) can thus be thought of as a collection of named relation schemas.

In implementations, the domain of each attribute is effectively a data type and a named relation schema is effectively a relation variable or relvar for short (see **Relation Variables** below).

In SQL, a database language for relational databases, relations are represented by tables, where each row of a table represents a single tuple, and where the values of each attribute form a column.

Examples

Below is an example of a relation having three named attributes: 'ID' from the domain of integers, and 'Name' and 'Address' from the domain of strings:

ID (Integer)	Name (String)	Address (String)
102	Yonezawa Akinori	Naha, Okinawa
202	Murata Makoto	Sendai, Miyagi
104	Sakamura Ken	Kumamoto, Kumamoto
152	Matsumoto Yukihiro	Okinawa, Okinawa

A predicate for this relation, using the attribute names to denote free variables, might be "Employee number *ID* is known as *Name* and lives at *Address*". Examination of the relation tells us that there are just four tuples for which the predicate holds true. So, for example, employee 102 is known only by that name, Yonezawa Akinori, and does not live anywhere else but in Naha, Okinawa. Also, apart from the four employees shown,there is no other employee who has both a name and an address.

Under the definition of **body**, the tuples of a body do not appear in any particular order - one cannot say "The tuple of 'Murata Makoto' is above the tuple of 'Matsumoto Yukihiro'", nor can one say "The tuple of 'Yonezawa Akinori' is the first tuple." A similar comment applies to the rows of an SQL table.

Under the definition of **heading** the elements of a element do not appear in any particular order either, nor, therefore do the elements of a tuple. A similar comment does *not* apply here to SQL, which does define an ordering to the columns of a table.

Relation Variables

A relational database consists of named relation variables (relvars) for the purposes of updating the database in response to changes in the real world. An update to a single relvar causes the body of the relation assigned to that variable to be replaced by a different set of tuples. Such variables are classified into two classes: **base relation variables** and **derived relation variables**, the latter also known as **virtual relvars** but usually referred to by the short term **view**.

A **base relation variable** is a relation variable which is not derived from any other relation variables. In SQL the term **base table** equates approximately to base relation variable.

A view can be defined by an expression using the operators of the relational algebra or the relational calculus. Such an expression operates on one or more relations and when evaluated yields another relation. The result is sometimes referred to as a "derived" relation when the operands are relations assigned to database variables. A view is defined by giving a name to such an expression, such that the name can subsequently be used as a variable name. (Note that the expression must then mention at least one base relation variable.)

By using a Data Definition Language (DDL), it is able to define base relation variables. In SQL, CREATE TABLE syntax is used to define base tables. The following is an example.

```
CREATE TABLE List_of_people (
 ID INTEGER,
 Name CHAR(40),
 Address CHAR(200),
 PRIMARY KEY (ID)
)
```

The Data Definition Language (DDL) is also used to define derived relation variables. In SQL, CREATE VIEW syntax is used to define a derived relation variable. The following is an example.

```
CREATE VIEW List_of_Okinawa_people AS (
 SELECT ID, Name, Address
```

```
  FROM List_of_people
  WHERE Address LIKE '%, Okinawa'
)
```

References

€ Date, C. J. (2004). *An Introduction to Database Systems* (8 ed.). Addison€Wesley. ISBN∫0-321-19784-4.

Table (database)

In relational databases and flat file databases, a **table** is a set of data elements (values) that is organized using a model of vertical columns (which are identified by their name) and horizontal rows, the cell being the unit where a row and column intersect. A table has a specified number of columns, but can have any number of rows [*citation needed*]. Each row is identified by the values appearing in a particular column subset which has been identified as a unique key index.

Table is another term for relations; although there is the difference in that a table is usually a multiset (bag) of rows whereas a relation is a set and does not allow duplicates. Besides the actual data rows, tables generally have associated with them some metadata, such as constraints on the table or on the values within particular columns.Wikipedia:Disputed statement|

The data in a table does not have to be physically stored in the database. Views are also relational tables, but their data are calculated at query time. Another example are nicknames[*clarify*], which represent a pointer to a table in another database.

Comparisons with other data structures

In non-relational systems, hierarchical databases, the distant counterpart of a table is a structured file, representing the rows of a table in each record of the file and each column in a record.This structure implies that a record can have repeating information, Generally in the child data segments.Data are stored in sequence of records which are equivalent to table term of a relational database.with each record having equivalent rows.

Unlike a spreadsheet, the datatype of field is ordinarily defined by the schema describing the table. Some SQL systems, such as SQLite, are less strict about field datatype definitions.

Tables versus relations

In terms of the relational model of databases, a table can be considered a convenient representation of a relation, but the two are not strictly equivalent. For instance, an SQL table can potentially contain duplicate rows, whereas a true relation cannot contain duplicate tuples. Similarly, representation as a table implies a particular ordering to the rows and columns, whereas a relation is explicitly unordered. However, the database system does not guarantee any ordering of the rows unless an ORDER BY clause is specified in the SELECT statement that queries the table.

An equally valid representations of a relation is as an *n*-dimensional chart, where *n* is the number of attributes (a table's columns). For example, a relation with two attributes and three values can be represented as a table with two columns and three rows, or as a two-dimensional graph with three points. The table and graph representations are only equivalent if the ordering of rows is not significant, and the table has no duplicate rows.

Table types

Two types of tables exist:

- A relational table, which is the basic structure to hold user data in a relational database.
- An object table, which is a table that uses an object type to define a column. It is defined to hold instances of objects of a defined type.

In SQL, the CREATE TABLE statement creates these tables.

References

Column (database)

In the context of a relational database table, a **column** is a set of data values of a particular simple type, one for each row of the table.[1] The columns provide the structure according to which the rows are composed.

The term *field* is often used interchangeably with *column*, although many consider it more correct to use *field* (or *field value*) to refer specifically to the single item that exists at the intersection between one row and one column.

In relational database terminology, column's equivalent is called **attribute**.

For example, a table that represents companies might have the following columns:

- ID (integer identifier, unique to each row)
- Name (text)
- Address line 1 (text)
- Address line 2 (text)
- City (integer identifier, drawn from a separate table of cities, from which any state or country information would be drawn)
- Postal code (text)
- Industry (integer identifier, drawn from a separate table of industries)
- etc.

Each row would provide a data value for each column and would then be understood as a single structured data value, in this case representing a company. More formally, each row can be interpreted as a relvar, composed of a set of tuples, with each tuple consisting of the two items: the name of the relevant column and the value this row provides for that column.

	Column 1	**Column 2**
Row 1	Row 1, Column 1	Row 1, Column 2
Row 2	Row 2, Column 1	Row 2, Column 2
Row 3	Row 3, Column 1	Row 3, Column 2

Examples of database: MySQL, SQL Server, Access, Oracle, Sybase, DB2.

Coding involved: SQL [Structured Query Language]

See more at SQL.

References

[1] The term "column" also has equivalent application in other, more generic contexts. See e.g., Flat file database, Table (information).

Row (database)

In the context of a relational database, a **row** *f* also called a **record** or **tuple** *f* represents a single, implicitly structured data item in a table. In simple terms, a database table can be thought of as consisting of *rows* and columns or fields. Each row in a table represents a set of related data, and every row in the table has the same structure.

For example, in a table that represents companies, each row would represent a single company. Columns might represent things like company name, company street address, whether the company is publicly held, its VAT number, etc.. In a table that represents *the association* of employees with departments, each row would associate one employee with one department.

In a less formal usage, e.g. for a database which is not formally relational, a record is equivalent to a row as described above, but is not usually referred to as a row.

The implicit structure of a row, and the meaning of the data values in a row, requires that the row be understood as providing a succession of data values, one in each column of the table. The row is then interpreted as a relvar composed of a set of tuples, with each tuple consisting of the two items: the name of the relevant column and the value this row provides for that column.

Each column expects a data value of a particular type. For example, one column might require a unique identifier, another might require text representing a person's name, another might require an integer representing hourly pay in cents.

-	**Column 1**	**Column 2**
Row 1	Row 1, Column 1	Row 1, Column 2
Row 2	Row 2, Column 1	Row 2, Column 2
Row 3	Row 3, Column 1	Row 3, Column 2

View (SQL)

In database theory, a **view** is the result set of a *stored* query *f* or map-and-reduce functions *f* on the data, which the database users can query just as they would a persistent database collection object. This pre-established query command is kept in the database dictionary. Unlike ordinary *base tables* in a relational database, a view does not form part of the physical schema: as a result set, it is a virtual table computed or collated from data in the database, dynamically when access to that view is requested. Changes applied to the data in a relevant *underlying table* are reflected in the data shown in subsequent invocations of the view. In some NoSQL databases, views are the only way to query data.

Views can provide advantages over tables:

- Views can represent a subset of the data contained in a table; consequently, a view can limit the degree of exposure of the underlying tables to the outer world: a given user may have permission to query the view, while denied access to the rest of the base table.
- Views can join and simplify multiple tables into a single virtual table
- Views can act as aggregated tables, where the database engine aggregates data (sum, average etc.) and presents the calculated results as part of the data
- Views can hide the complexity of data; for example a view could appear as Sales2000 or Sales2001, transparently partitioning the actual underlying table
- Views take very little space to store; the database contains only the definition of a view, not a copy of all the data which it presents
- Depending on the SQL engine used, views can provide extra security

Just as a function (in programming) can provide abstraction, so can a database view. In another parallel with functions, database users can manipulate nested views, thus one view can aggregate data from other views. Without the use of views, the normalization of databases above second normal form would become much more difficult. Views can make it easier to create lossless join decomposition.

Just as rows in a base table lack any defined ordering, rows available through a view do not appear with any default sorting. A view is a relational table, and the relational model defines a table as a set of rows. Since sets are not ordered - by definition - nor are the rows of a view. Therefore, an ORDER BY clause in the view definition is meaningless; the SQL standard (SQL:2003) does not allow an ORDER BY clause in the subquery of a CREATE VIEW command, just as it is refused in a CREATE TABLE statement. However, sorted data can be obtained from a view, in the same way as any other table *f* as part of a query statement on that view. Nevertheless, some DBMS (such as Oracle Database) do not abide by this SQL standard restriction.

Read-only vs. updatable views

Database practitioners can define views as read-only or updatable. If the database system can determine the reverse mapping from the view schema to the schema of the underlying base tables, then the view is updatable. INSERT, UPDATE, and DELETE operations can be performed on updatable views. Read-only views do not support such operations because the DBMS cannot map the changes to the underlying base tables. A view update is done by key preservation.

Some systems support the definition of INSTEAD OF triggers on views. This technique allows the definition of other logic for execution in place of an insert, update, or delete operation on the views. Thus database systems can implement data modifications based on read-only views. However, an INSTEAD OF trigger does not change the read-only or updatable property of the view itself.

Advanced view features

Various database management systems have extended the views from read-only subsets of data.

Oracle Database introduced the concept of materialized views: pre-executed, non-virtual views commonly used in data warehousing. They give a static snapshot of the data and may include data from remote sources. The accuracy of a materialized view depends on the frequency of trigger mechanisms behind its updates. IBM DB2 provides so-called "materialized query tables" (MQTs) for the same purpose. Microsoft SQL Server introduced in its 2000 version indexed views which only store a separate index from the table, but not the entire data. PostgreSQL implemented materialized views in its 9.3 release.

Equivalence

A view is equivalent to its source query. When queries are run against views, the query is modified. For example, if there exists a view named accounts_view with the content as follows:

```
accounts_view:
--------------
SELECT name,
       money_received,
       money_sent,
       (money_received - money_sent) AS balance,
       address,
         ...
  FROM table_customers c
  JOIN accounts_table a
    ON a.customer_id = c.customer_id
```

then the application could run a simple query such as:

```
Simple query
------------
SELECT name,
       balance
  FROM accounts_view
```

The RDBMS then takes the simple query, replaces the equivalent view, then sends the following to the query optimizer:

```
Preprocessed query:
-------------------
SELECT name,
       balance
  FROM (SELECT name,
               money_received,
               money_sent,
               (money_received - money_sent) AS balance,
               address,
                      ...
          FROM table_customers c JOIN accounts_table a
               ON a.customer_id = c.customer_id        )
```

From this point on the optimizer takes the query, removes unnecessary complexity (for example: it is not necessary to read the address, since the parent invocation does not make use of it) and then sends the query to the SQL engine for processing.

External links

- Materialized query tables in DB2 [1]
- Views in Microsoft SQL Server [2]
- Views in MySQL [3]
- Views in PostgreSQL [4]
- Views in SQLite [5]
- Views in Oracle 11.2 [6]
- Views in CouchDB [7]
- Materialized Views in Oracle 11.2 [8]

References

[1] http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=/com.ibm.db2z10.doc.intro/src/tpc/db2z_typesoftables.htm
[2] http://msdn.microsoft.com/en-us/library/ms187956.aspx
[3] http://dev.mysql.com/doc/refman/5.1/en/views.html
[4] http://www.postgresql.org/docs/current/interactive/tutorial-views.html
[5] http://www.sqlite.org/lang_createview.html
[6] http://download.oracle.com/docs/cd/E11882_01/server.112/e17118/statements_8004.htm#SQLRF01504
[7] http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views
[8] http://download.oracle.com/docs/cd/E11882_01/server.112/e17118/statements_6002.htm#SQLRF01302

Database transaction

A **transaction** comprises a unit of work performed within a database management system (or similar system) against a database, and treated in a coherent and reliable way independent of other transactions. Transactions in a database environment have two main purposes:

1. To provide reliable units of work that allow correct recovery from failures and keep a database consistent even in cases of system failure, when execution stops (completely or partially) and many operations upon a database remain uncompleted, with unclear status.
2. To provide isolation between programs accessing a database concurrently. If this isolation is not provided, the program's outcome are possibly erroneous.

A database transaction, by definition, must be atomic, consistent, isolated and durable.[1] Database practitioners often refer to these properties of database transactions using the acronym ACID.

Transactions provide an "all-or-nothing" proposition, stating that each work-unit performed in a database must either complete in its entirety or have no effect whatsoever. Further, the system must isolate each transaction from other transactions, results must conform to existing constraints in the database, and transactions that complete successfully must get written to durable storage.

Purpose

Databases and other data stores which treat the integrity of data as paramount often include the ability to handle transactions to maintain the integrity of data. A single transaction consists of one or more independent units of work, each reading and/or writing information to a database or other data store. When this happens it is often important to ensure that all such processing leaves the database or data store in a consistent state.

Examples from double-entry accounting systems often illustrate the concept of transactions. In double-entry accounting every debit requires the recording of an associated credit. If one writes a check for $100 to buy groceries, a transactional double-entry accounting system must record the following two entries to cover the single transaction:

1. Debit $100 to Groceries Expense Account
2. Credit $100 to Checking Account

A transactional system would make both entries pass or both entries would fail. By treating the recording of multiple entries as an atomic transactional unit of work the system maintains the integrity of the data recorded. In other words, nobody ends up with a situation in which a debit is recorded but no associated credit is recorded, or vice versa.

Transactional databases

A **transactional database** is a DBMS where write transactions on the database are able to be rolled back if they are not completed properly (e.g. due to power or connectivity loss).

Most modern[2] relational database management systems fall into the category of databases that support transactions.

In a database system a transaction might consist of one or more data-manipulation statements and queries, each reading and/or writing information in the database. Users of database systems consider consistency and integrity of data as highly important. A simple transaction is usually issued to the database system in a language like SQL wrapped in a transaction, using a pattern similar to the following:

1. Begin the transaction
2. Execute a set of data manipulations and/or queries
3. If no errors occur then commit the transaction and end it
4. If errors occur then rollback the transaction and end it

If no errors occurred during the execution of the transaction then the system commits the transaction. A transaction commit operation applies all data manipulations within the scope of the transaction and persists the results to the database. If an error occurs during the transaction, or if the user specifies a rollback operation, the data manipulations within the transaction are not persisted to the database. In no case can a partial transaction be committed to the database since that would leave the database in an inconsistent state.

Internally, multi-user databases store and process transactions, often by using a transaction ID or XID.

There are multiple varying ways for transactions to be implemented other than the simple way documented above. Nested transactions, for example, are transactions which contain statements within them that start new transactions (i.e. sub-transactions). Multi-level transactions are similar but have a few extra properties[*citation needed*]. Another type of transaction is the compensating transaction.

In SQL

SQL is inherently transactional, and a transaction is automatically started when another ends. Some databases extend SQL and implement a START TRANSACTION statement, but while seemingly signifying the start of the transaction it merely deactivates autocommit.[*citation needed*]

The result of any work done after this point will remain invisible to other database-users until the system processes a COMMIT statement. A ROLLBACK statement can also occur, which will undo any work performed since the last transaction. Both COMMIT and ROLLBACK will end the transaction, and start anew. If autocommit was disabled using START TRANSACTION, autocommit will often also be reenabled.

Some database systems allow the synonyms BEGIN, BEGIN WORK and BEGIN TRANSACTION, and may have other options available.

Distributed transactions

Database systems implement distributed transactions as transactions against multiple applications or hosts. A distributed transaction enforces the ACID properties over multiple systems or data stores, and might include systems such as databases, file systems, messaging systems, and other applications. In a distributed transaction a coordinating service ensures that all parts of the transaction are applied to all relevant systems. As with database and other transactions, if any part of the transaction fails, the entire transaction is rolled back across all affected systems.

Transactional filesystems

The Namesys Reiser4 filesystem for Linux[3] supports transactions, and as of Microsoft Windows Vista, the Microsoft NTFS filesystem[4] supports distributed transactions across networks.

References

[1] A transaction is a group of operations that are atomic, consistent, isolated, and durable ([[ACID (http://msdn.microsoft.com/en-us/library/aa366402(VS.85).aspx)]).]
[2] http://en.wikipedia.org/w/index.php?title=Database_transaction&action=edit
[3] http://namesys.com/v4/v4.html#committing
[4] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/fs/portal.asp

Further reading

€ Philip A. Bernstein, Eric Newcomer (2009): *Principles of Transaction Processing*, 2nd Edition (http://www.elsevierdirect.com/product.jsp?isbn=9781558606234), Morgan Kaufmann (Elsevier), ISBN 978-1-55860-623-4
€ Gerhard Weikum, Gottfried Vossen (2001), *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*, Morgan Kaufmann, ISBN 1-55860-508-8

Transaction log

In the field of databases in computer science, a **transaction log** (also **transaction journal**, **database log**, **binary log** or **audit trail**) is a history of actions executed by a database management system to guarantee ACID properties over crashes or hardware failures. Physically, a log is a file of updates done to the database, stored in stable storage.

If, after a start, the database is found in an inconsistent state or not been shut down properly, the database management system reviews the database logs for uncommitted transactions and rolls back the changes made by these transactions. Additionally, all transactions that are already committed but whose changes were not yet materialized in the database are re-applied. Both are done to ensure atomicity and durability of transactions.

This term is not to be confused with other, human-readable logs that a database management system usually provides.

Anatomy of a general database log

A database log record is made up of:

- *Log Sequence Number*: A unique id for a log record. With LSNs, logs can be recovered in constant time. Most logs' LSNs are assigned in monotonically increasing order, which is useful in recovery algorithms, like ARIES.
- *Prev LSN*: A link to the last log record. This implies database logs are constructed in linked list form.
- *Transaction ID number*: A reference to the database transaction generating the log record.
- *Type*: Describes the type of database log record.
- Information about the actual changes that triggered the log record to be written.

Types of database log records

All log records include the general log attributes above, and also other attributes depending on their type (which is recorded in the *Type* attribute, as above).

- **Update Log Record** notes an update (change) to the database. It includes this extra information:
  - *PageID*: A reference to the Page ID of the modified page.
  - *Length and Offset*: Length in bytes and offset of the page are usually included.
  - *Before and After Images*: Includes the value of the bytes of page before and after the page change. Some databases may have logs which include one or both images.
- **Compensation Log Record** notes the rollback of a particular change to the database. Each correspond with exactly one other Update Log Record (although the corresponding update log record is not typically stored in the Compensation Log Record). It includes this extra information:
  - *undoNextLSN*: This field contains the LSN of the next log record that is to be undone for transaction that wrote the last Update Log.
- **Commit Record** notes a decision to commit a transaction.
- **Abort Record** notes a decision to abort and hence roll back a transaction.
- **Checkpoint Record** notes that a checkpoint has been made. These are used to speed up recovery. They record information that eliminates the need to read a long way into the log's past. This varies according to checkpoint algorithm. If all dirty pages are flushed while creating the checkpoint (as in PostgreSQL), it might contain:
  - *redoLSN*: This is a reference to the first log record that corresponds to a dirty page. i.e. the first update that wasn't flushed at checkpoint time. This is where redo must begin on recovery.
  - *undoLSN*: This is a reference to the oldest log record of the oldest in-progress transaction. This is the oldest log record needed to undo all in-progress transactions.

- **Completion Record** notes that all work has been done for this particular transaction. (It has been fully committed or aborted)

Tables

These tables are maintained in memory, and can be efficiently reconstructed (if not exactly, to an equivalent state) from the log and the database:

- **Transaction Table**: The table contains one entry for each active transaction. This includes Transaction ID and lastLSN, where lastLSN describes the LSN of the most recent log record for the transaction.
- **Dirty Page Table**: The table contains one entry for each dirty page that hasn't been written to disk. The entry contains recLSN, where recLSN is the LSN of the first log record that caused the page to be dirty.
- **Transaction Log**: A DBMS uses a transaction log to keep track of all transactions that updates the database. The information stored in this log is used by DBMS for a recovery requirement triggered by 'Roll Back' statement.

Database trigger

A **database trigger** is procedural code that is automatically executed in response to certain events on a particular table or view in a database. The trigger is mostly used for maintaining the integrity of the information on the database. For example, when a new record (representing a new worker) is added to the employees table, new records should also be created in the tables of the taxes, vacations and salaries.

The need and the usage

Triggers are commonly used to:

- audit changes (e.g. keep a log of the users and roles involved in changes)
- enhance changes (e.g. ensure that every change to a record is time-stamped by the server's clock)
- enforce business rules (e.g. require that every invoice have at least one line item)
- execute business rules (e.g. notify a manager every time an employee's bank account number changes)
- replicate data (e.g. store a record of every change, to be shipped to another database later)
- enhance performance (e.g. update the account balance after every detail transaction, for faster queries)

The examples above are called Data Manipulation Language (DML) triggers because the triggers are defined as part of the Data Manipulation Language and are executed at the time the data is manipulated. SomeWikipedia:Avoid weasel words systems also support non-data triggers, which fire in response to Data Definition Language (DDL) events such as creating tables, or runtime events such as logon, commit and rollback. Such DDL triggers can be used for database auditing purposes.

The following are major features of database triggers and their effects:

- triggers do not accept parameters or arguments (but may store affected-data in temporary tables)
- triggers cannot perform commit or rollback operations because they are part of the triggering SQL statement (only through autonomous transactions)

Triggers in DBMS

Below follows a series of descriptions of how some popular DBMS support triggers.

Oracle

In addition to triggers that fire when data is modified, Oracle 9i supports triggers that fire when schema objects (that is, tables) are modified and when user logon or logoff events occur. These trigger types are referred to as "Schema-level triggers".

Schema-level triggers

- After Creation
- Before Alter
- After Alter
- Before Drop
- After Drop
- Before Logoff
- After Logon

The four main types of triggers are:

1. Row Level Trigger: This gets executed before or after *any column value of a row* changes
2. Column Level Trigger: This gets executed before or after the *specified column* changes
3. For Each Row Type: This trigger gets executed once for each row of the result set caused by insert/update/delete
4. For Each Statement Type: This trigger gets executed only once for the entire result set, but fires each time the statement is executed.

Mutating tables

When a single SQL statement modifies several rows of a table at once, the order of the operations is not well-defined; there is no "order by" clause on "update" statements, for example. Row-level triggers are executed as each row is modified, so the order in which trigger code is run is also not well-defined. Oracle protects the programmer from this uncertainty by preventing row-level triggers from modifying other rows in the same table€ this is the "mutating table" in the error message. Side-effects on other tables are allowed, however.

One solution is to have row-level triggers place information into a temporary table indicating what further changes need to be made, and then have a statement-level trigger fire just once, at the end, to perform the requested changes and clean up the temporary table.

Because a foreign key's referential actions are implemented via implied triggers, they are similarly restricted. This may become a problem when defining a self-referential foreign key, or a cyclical set of such constraints, or some other combination of triggers and CASCADE rules, e.g. user deletes a record from table A, CASCADE rule on table A deletes a record from table B, trigger on table B attempts to SELECT from table A, error occurs.

Microsoft SQL Server

Microsoft SQL Server supports triggers either before, after, or instead of an insert, update or delete operation. They can be set on tables and views with the constraint that a view can be referenced only by an INSTEAD OF trigger.

Microsoft SQL Server 2005 introduced support for Data Definition Language (DDL) triggers, which can fire in reaction to a very wide range of events, including:

- Drop table
- Create table
- Alter table
- Login events

A full list [1] is available on MSDN.

Performing conditional actions in triggers (or testing data following modification) is done through accessing the temporary *Inserted* and *Deleted* tables.

PostgreSQL

PostgreSQL introduced support for triggers in 1997. The following functionality in SQL:2003 was previously not implemented in PostgreSQL:

- SQL allows triggers to fire on updates to specific columns; As of version 9.0 of PostgreSQL this feature is also implemented in PostgreSQL.
- The standard allows the execution of a number of SQL statements other than SELECT, INSERT, UPDATE, such as CREATE TABLE as the triggered action. This can be done through creating a stored procedure or function to call CREATE TABLE.[2]

Synopsis:

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
    ON TABLE [ FOR [ EACH ] { ROW | STATEMENT } ]
    EXECUTE PROCEDURE funcname ( arguments )
```

Firebird

Firebird supports multiple row-level, BEFORE or AFTER, INSERT, UPDATE, DELETE (or any combination thereof) triggers per table, where they are always "in addition to" the default table changes, and the order of the triggers relative to each other can be specified where it would otherwise be ambiguous (POSITION clause.) Triggers may also exist on views, where they are always "instead of" triggers, replacing the default updatable view logic. (Before version 2.1, triggers on views deemed updatable would run in addition to the default logic.)

Firebird does not raise mutating table exceptions (like Oracle), and triggers will by default both nest and recurse as required (SQL Server allows nesting but not recursion, by default.) Firebird's triggers use NEW and OLD context variables (not Inserted and Deleted tables,) and provide UPDATING, INSERTING, and DELETING flags to indicate the current usage of the trigger.

```
{CREATE | RECREATE | CREATE OR ALTER} TRIGGER name FOR {table name |
view name}
 [ACTIVE | INACTIVE]
 {BEFORE | AFTER}
 {INSERT [OR UPDATE] [OR DELETE] | UPDATE [OR INSERT] [OR DELETE] |
DELETE [OR UPDATE] [OR INSERT] }
 [POSITION n] AS
BEGIN
```

```
  .....
END
```

As of version 2.1, Firebird additionally supports the following database-level triggers:

- CONNECT (exceptions raised here prevent the connection from completing)
- DISCONNECT
- TRANSACTION START
- TRANSACTION COMMIT (exceptions raised here prevent the transaction from committing, or preparing if a two-phase commit is involved)
- TRANSACTION ROLLBACK

Database-level triggers can help enforce multi-table constraints, or emulate materialized views. If an exception is raised in a TRANSACTION COMMIT trigger, the changes made by the trigger so far are rolled back and the client application is notified, but the transaction remains active as if COMMIT had never been requested; the client application can continue to make changes and re-request COMMIT.

Syntax for database triggers:

```
{CREATE | RECREATE | CREATE OR ALTER} TRIGGER name
  [ACTIVE | INACTIVE] ON
  {CONNECT | DISCONNECT | TRANSACTION START | TRANSACTION COMMIT |
TRANSACTION ROLLBACK}
  [POSITION n] AS
BEGIN
  .....
END
```

MySQL

MySQL 5.0.2 introduced support for triggers. MySQL supports these trigger types:

- Insert Trigger
- Update Trigger
- Delete Trigger

Note: MySQL allows only one trigger of each type on each table (i.e. one before insert, one after insert, one before update, one after update, one before delete and one after delete).

Note: MySQL does NOT fire triggers outside of a statement (i.e. API's, foreign key cascades)

The SQL:2003 standard mandates that triggers give programmers access to record variables by means of a syntax such as REFERENCING NEW AS n. For example, if a trigger is monitoring for changes to a salary column one could write a trigger like the following:

```
CREATE TRIGGER salary_trigger
    BEFORE UPDATE ON employee_table
    REFERENCING NEW ROW AS n, OLD ROW AS o
    FOR EACH ROW
    IF n.salary <> o.salary THEN

    END IF;
;
```

Sample Mytrigger as follows:

```
-- First of all, drop any other trigger with the same name
DROP TRIGGER IF EXISTS `Mytrigger`;
-- Create New Trigger
DELIMITER $$

CREATE
    /*[DEFINER = { user | CURRENT_USER }]*/
    TRIGGER `DB`.`mytriggers` BEFORE/AFTER INSERT/UPDATE/DELETE
    ON `DB`.`<Table Name>`
    FOR EACH ROW BEGIN

    END$$

DELIMITER ;

-- Example:
DROP TRIGGER IF EXISTS `Mytrigger`;

DELIMITER $$
CREATE TRIGGER `Mytrigger`
AFTER INSERT ON Table_Current
FOR EACH ROW
BEGIN

    UPDATE Table_Record

    SET `Value` = NEW.`Value`
    WHERE `Name` = NEW.`Name`
    AND `Value` < NEW.`Value`;

END $$
DELIMITER;
```

IBM DB2 LUW

IBM DB2 for distributed systems known as DB2 for LUW (LUW means Linux Unix Windows) supports three trigger types: Before trigger, After trigger and Instead of trigger. Both statement level and row level triggers are supported. If there are more triggers for same operation on table then firing order is determined by trigger creation data. Since version 9.7 IBM DB2 supports autonomous transactions [3].

Before trigger is for checking data and deciding if operation should be permitted. If exception is thrown from before trigger then operation is aborted and no data are changed. In DB2 before triggers are read only*f* you can't modify data in before triggers. After triggers are designed for post processing after requested change was performed. After triggers can write data into tables and unlike someWikipedia:Avoid weasel words other databases you can write into any table including table on which trigger operates. Instead of triggers are for making views writeable.

Triggers are usually programmed in SQL PL language.

SQLite

```
CREATE [TEMP | TEMPORARY] TRIGGER [IF NOT EXISTS] [database_name .]
trigger_name
[BEFORE | AFTER | INSTEAD OF] {DELETE | INSERT | UPDATE [OF column_name
 [, column_name]...]}
ON {table_name | view_name}
   [FOR EACH ROW] [WHEN condition]
BEGIN
   ...
END
```

SQLite only supports row-level triggers, not statement-level triggers.

Updateable views, which are not supported in SQLite, can be emulated with INSTEAD OF triggers.

XML databases

An example of implementation of triggers in non-relational database can be Sedna, that provides support for triggers based on XQuery. Triggers in Sedna were designed to be analogous to SQL:2003 triggers, but natively base on XML query and update languages (XPath, XQuery and XML update language).

A trigger in Sedna is set on any nodes of an XML document stored in database. When these nodes are updated, the trigger automatically executes XQuery queries and updates specified in its body. For example, the following trigger cancels person node deletion if there are any open auctions referenced by this person:

```
CREATE TRIGGER "trigger3"
    BEFORE DELETE
    ON doc("auction")/site//person
    FOR EACH NODE
    DO
    {
       if(exists($WHERE//open_auction/bidder/personref/@person=$OLD/@id))
       then ( )
       else $OLD;
    }
```

References

[1] http://msdn2.microsoft.com/en-us/library/ms189871(SQL.90).aspx
[2] http://www.postgresql.org/docs/9.0/static/sql-createtrigger.html
[3] http://www.ibm.com/developerworks/data/library/techarticle/dm-0907autonomoustransactions/index.html

External links

- Microsoft SQL Server DROP TRIGGER (http://msdn2.microsoft.com/en-us/library/aa258846(SQL.80).aspx)
- MySQL Database triggers (http://dev.mysql.com/doc/refman/5.0/en/triggers.html)
- MySQL DB Create Triggers (http://dev.mysql.com/doc/refman/5.0/en/create-trigger.html)
- DB2 CREATE TRIGGER statement (http://publib.boulder.ibm.com/infocenter/db2luw/v9/topic/com.ibm.db2.udb.admin.doc/doc/r0000931.htm)
- Oracle CREATE TRIGGER (http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_7004.htm#sthref7885)

- PostgreSQL CREATE TRIGGER (http://www.postgresql.org/docs/8.2/static/sql-createtrigger.html)
- Oracle Mutating Table Problems with DELETE CASCADE (http://www.akadia.com/services/ora_mutating_table_problems.html)
- SQLite Query Language: CREATE TRIGGER (http://www.sqlite.org/lang_createtrigger.html)

Database index

A **database index** is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and the use of more storage space to maintain the extra copy of data. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

In a relational database, indexes are used to quickly and efficiently provide the exact location of the corresponding data. An index is a copy of select columns of data from a table that can be searched very efficiently that also includes a low level disk block address or direct link to the complete row of data it was copied from. Some databases extend the power of indexing by allowing indices to be created on functions or expressions. For example, an index could be created on `upper(last_name)`, which would only store the upper case versions of the last_name field in the index. Another option sometimes supported is the use of "filtered" indices, where index entries are created only for those records that satisfy some conditional expression. A further aspect of flexibility is to permit indexing on user-defined functions, as well as expressions formed from an assortment of built-in functions.

Usage

Support for fast lookup

Most database software includes indexing technology that enables sub-linear time lookup to improve performance, as linear search is inefficient for large databases.

Suppose a database contains N data items and it is desired to retrieve one or two of them based on the value of one of the fields. A naive implementation would retrieve and examine each item until a match was not found. A successful lookup would retrieve half the objects on average; an unsuccessful lookup all of them for each attempt. This means that the number of operations in the worst case is O(N) or linear time. Since databases commonly contain millions of objects and since lookup is a common operation, it is often desirable to improve on this performance.

An index is any data structure that improves the performance of lookup. There are many different data structures used for this purpose, and in fact a substantial proportion of the field of Computer Science is devoted to the design and analysis of index data structures. [*citation needed*] There are complex design trade-offs involving lookup performance, index size, and index update performance. Many index designs exhibit logarithmic (O(log(N))) lookup performance and in some applications it is possible to achieve flat (O(1)) performance.

Policing the database constraints

Indices are used to police database constraints, such as UNIQUE, EXCLUSION, PRIMARY KEY and FOREIGN KEY. An index may be declared as UNIQUE which creates an implicit constraint on the underlying table. Database systems usually implicitly create an index on a set of columns declared PRIMARY KEY, and some are capable of using an already existing index to police this constraint. Many database systems require that both referencing and referenced sets of columns in a FOREIGN KEY constraint are indexed, thus improving performance of inserts, updates and deletes to the tables participating in the constraint.

Some database systems support EXCLUSION constraint which ensures that for a newly inserted or updated record a certain predicate would hold for no other record. This may be used to implement a UNIQUE constraint (with equality predicate) or more complex constraints, like ensuring that no overlapping time ranges or no intersecting geometry objects would be stored in the table. An index supporting fast searching for records satisfying the predicate is required to police such a constraint.[1]

Index architecture

Non-clustered

The data is present in arbitrary order, but the **logical ordering** is specified by the index. The data rows may be spread throughout the table regardless of the value of the indexed column or expression. The non-clustered index tree contains the index keys in sorted order, with the leaf level of the index containing the pointer to the record (page and the row number in the data page in page-organized engines; row offset in file-organized engines).

In a non-clustered index:

- The physical order of the rows is not the same as the index order.
- Typically created on non-primary key columns used in JOIN, WHERE, and ORDER BY clauses.

There can be more than one non-clustered index on a database table.

Clustered

Clustering alters the data block into a certain distinct order to match the index, resulting in the row data being stored in order. Therefore, only one clustered index can be created on a given database table. Clustered indices can greatly increase overall speed of retrieval, but usually only where the data is accessed sequentially in the same or reverse order of the clustered index, or when a range of items is selected.

Since the physical records are in this sort order on disk, the next row item in the sequence is immediately before or after the last one, and so fewer data block reads are required. The primary feature of a clustered index is therefore the ordering of the physical data rows in accordance with the index blocks that point to them. Some databases separate the data and index blocks into separate files, others put two completely different data blocks within the same physical file(s).

Cluster

When multiple databases and multiple tables are joined, it's referred to as a **cluster** (not to be confused with clustered index described above). The records for the tables sharing the value of a cluster key shall be stored together in the same or nearby data blocks. This may improve the joins of these tables on the cluster key, since the matching records are stored together and less I/O is required to locate them.[2] The data layout in the tables which are parts of the cluster is defined by the cluster configuration. A cluster can be keyed with a B-Tree index or a hash table. The data block in which the table record will be stored is defined by the value of the cluster key.

Column order

The order in which columns are listed in the index definition is important. It is possible to retrieve a set of row identifiers using only the first indexed column. However, it is not possible or efficient (on most databases) to retrieve the set of row identifiers using only the second or greater indexed column.

For example, imagine a phone book that is organized by city first, then by last name, and then by first name. If you are given the city, you can easily extract the list of all phone numbers for that city. However, in this phone book it would be very tedious to find all the phone numbers for a given last name. You would have to look within each city's section for the entries with that last name. Some databases can do this, others just won,t use the index.

Applications and limitations

Indices are useful for many applications but come with some limitations. Consider the following SQL statement: `SELECT first_name FROM people WHERE last_name = 'Smith';`. To process this statement without an index the database software must look at the last_name column on every row in the table (this is known as a full table scan). With an index the database simply follows the B-tree data structure until the Smith entry has been found; this is much less computationally expensive than a full table scan.

Consider this SQL statement: `SELECT email_address FROM customers WHERE email_address LIKE '%@yahoo.com';`. This query would yield an email address for every customer whose email address ends with "@yahoo.com", but even if the email_address column has been indexed the database must perform a full index scan. This is because the index is built with the assumption that words go from left to right. With a wildcard at the beginning of the search-term, the database software is unable to use the underlying B-tree data structure (in other words, the WHERE-clause is *not sargable*). This problem can be solved through the addition of another index created on `reverse(email_address)` and a SQL query like this: `SELECT email_address FROM customers WHERE reverse(email_address) LIKE reverse('%@yahoo.com');`. This puts the wild-card at the right-most part of the query (now moc.oohay@%) which the index on reverse(email_address) can satisfy.

Types of indexes

Bitmap index

A bitmap index is a special kind of index that stores the bulk of its data as bit arrays (bitmaps) and answers most queries by performing bitwise logical operations on these bitmaps. The most commonly used indexes, such as B+trees, are most efficient if the values they index do not repeat or repeat a smaller number of times. In contrast, the bitmap index is designed for cases where the values of a variable repeat very frequently. For example, the gender field in a customer database usually contains two distinct values: male or female. For such variables, the bitmap index can have a significant performance advantage over the commonly used trees.

Dense index

A dense index in databases is a file with pairs of keys and pointers for every record in the data file. Every key in this file is associated with a particular pointer to *a record* in the sorted data file. In clustered indices with duplicate keys, the dense index points *to the first record* with that key.[3]

Sparse index

A sparse index in databases is a file with pairs of keys and pointers for every block in the data file. Every key in this file is associated with a particular pointer *to the block* in the sorted data file. In clustered indices with duplicate keys, the sparse index points *to the lowest search key* in each block.

Reverse index

A reverse key index reverses the key value before entering it in the index. E.g., the value 24538 becomes 83542 in the index. Reversing the key value is particularly useful for indexing data such as sequence numbers, where new key values monotonically increase.

Index implementations

Indices can be implemented using a variety of data structures. Popular indices include balanced trees, B+ trees and hashes.

In Microsoft SQL Server, the leaf node of the clustered index corresponds to the actual data, not simply a pointer to data that resides elsewhere, as is the case with a non-clustered index. Each relation can have a single clustered index and many unclustered indices.

Index concurrency control

An index is typically being accessed concurrently by several transactions and processes, and thus needs concurrency control. While in principle indexes can utilize the common database concurrency control methods, specialized concurrency control methods for indexes exist, which are applied in conjunction with the common methods for a substantial performance gain.

Covering index

In most cases, an index is used to quickly locate the data record(s) from which the required data is read. In other words, the index is only used to locate data records in the table and not to return data.

A covering index is a special case where the index itself contains the required data field(s) and can return the data.

Consider the following table (other fields omitted):

ID	Name	Other Fields
12	Plug	...
13	Lamp	...
14	Fuse	...

To find the Name for ID 13, an index on (ID) will be useful, but the record must still be read to get the Name. However, an index on (ID, Name) contains the required data field and eliminates the need to look up the record.

A covering index can dramatically speed up data retrieval but may itself be large due to the additional keys, which slow down data insertion & update. To reduce such index size, some systems allow non-key fields to be included in the index. Non-key fields are not themselves part of the index ordering but only included at the leaf level, allowing for a covering index with less overall index size.

Standardization

There is no standard about creating indexes because the ISO SQL Standard does not cover physical aspects. Indexes are one of the physical parts of database conception among others like storage (tablespace or filegroups). RDBMS vendors all give a CREATE INDEX syntax with some specific options which depends on functionalities they provide to customers.

References

[1] PostgreSQL 9.1.2 Documentation: CREATE TABLE (http://www.postgresql.org/docs/9.1/static/sql-createtable.html)

[2] Overview of Clusters (http://download.oracle.com/docs/cd/B12037_01/server.101/b10743/schema.htm#sthref1069) Oracle ' Database Concepts 10g Release 1 (10.1)

[3] Database Systems: The Complete Book. Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer D. Widom

Stored procedure

A **stored procedure** is a subroutine available to applications that access a relational database system. A stored procedure (sometimes called a **proc**, **sproc**, **StoPro**, **StoredProc**, **sp** or **SP**) is actually stored in the database data dictionary.

Typical use for stored procedures include data validation (integrated into the database) or access control mechanisms. Furthermore, stored procedures can consolidate and centralize logic that was originally implemented in applications. Extensive or complex processing that requires execution of several SQL statements is moved into stored procedures, and all applications call the procedures. One can use nested stored procedures by executing one stored procedure from within another.

Stored procedures are similar to user-defined functions (UDFs). The major difference is that UDFs can be used like any other expression within SQL statements, whereas stored procedures must be invoked using the CALL statement.[1]

```
CALL procedure(...)
```

or

```
EXECUTE procedure(...)
```

Stored procedures may return result sets, *i.e.* the results of a SELECT statement. Such result sets can be processed using cursors, by other stored procedures, by associating a result set locator, or by applications. Stored procedures may also contain declared variables for processing data and cursors that allow it to loop through multiple rows in a table. Stored procedure flow control statements typically include IF, WHILE, LOOP, REPEAT, and CASE statements, and more. Stored procedures can receive variables, return results or modify variables and return them, depending on how and where the variable is declared.

Implementation

The exact and correct implementation of stored procedures varies from one database system to another. Most major database vendors support them in some form. Depending on the database system, stored procedures can be implemented in a variety of programming languages, for example SQL, Java, C, or C++. Stored procedures written in non-SQL programming languages may or may not execute SQL statements themselves.

The increasing adoption of stored procedures led to the introduction of procedural elements to the SQL language in the SQL:1999 and SQL:2003 standards in the part SQL/PSM. That made SQL an imperative programming language. Most database systems offer proprietary and vendor-specific extensions, exceeding SQL/PSM. A standard specification for Java stored procedures exists as well as SQL/JRT.

Database system	Implementation language
CUBRID	Java
DB2	SQL PL (close to the SQL/PSM standard) or Java
Firebird	PSQL (Fyracle also supports portions of Oracle's PL/SQL)
Informix	SPL or Java
Microsoft SQL Server	Transact-SQL and various .NET Framework languages
MySQL	own stored procedures, closely adhering to SQL/PSM standard.
Oracle	PL/SQL or Java
PostgreSQL	PL/pgSQL, can also use own function languages such as pl/perl or pl/php
Sybase ASE	Transact-SQL

Other uses

In some systems, stored procedures can be used to control transaction management; in others, stored procedures run inside a transaction such that transactions are effectively transparent to them. Stored procedures can also be invoked from a database trigger or a condition handler. For example, a stored procedure may be triggered by an insert on a specific table, or update of a specific field in a table, and the code inside the stored procedure would be executed. Writing stored procedures as condition handlers also allows database administrators to track errors in the system with greater detail by using stored procedures to catch the errors and record some audit information in the database or an external resource like a file.

Comparison with dynamic SQL

Overhead: Because stored procedure statements are stored directly in the database, they *may* remove all or part of the compilation overhead that is typically required in situations where software applications send inline (dynamic) SQL queries to a database. (However, most database systems implement "statement caches" and other mechanisms to avoid repetitive compilation of dynamic SQL statements.) In addition, while they avoid some overhead, pre-compiled SQL statements add to the complexity of creating an optimal execution plan because not all arguments of the SQL statement are supplied at compile time. Depending on the specific database implementation and configuration, mixed performance results will be seen from stored procedures versus generic queries or user defined functions.

Avoidance of network traffic: A major advantage with stored procedures is that they can run directly within the database engine. In a production system, this typically means that the procedures run entirely on a specialized database server, which has direct access to the data being accessed. The benefit here is that network communication costs can be avoided completely. This becomes particularly important for complex series of SQL statements.

Encapsulation of business logic: Stored procedures allow programmers to embed business logic as an API in the database, which can simplify data management and reduce the need to encode the logic elsewhere in client programs. This can result in a lesser likelihood of data corruption by faulty client programs. The database system can ensure data integrity and consistency with the help of stored procedures.

Delegation of access-rights: In many systems, stored procedures can be granted access rights to the database that users who execute those procedures do not directly have.

Some protection from SQL injection attacks: Stored procedures can be used to protect against injection attacks. Stored procedure parameters will be treated as data even if an attacker inserts SQL commands. Also, some DBMSs will check the parameter's type. A stored procedure that in turn generates dynamic SQL using the input is however

still vulnerable to SQL injections unless proper precautions are taken.

Comparison with functions

- A function is a subprogram written to perform certain computations
- A scalar function returns only a single value (or NULL), whereas a table function returns a (relational) table comprising zero or more rows, each row with one or more columns.
- Functions must return a value (using the RETURN keyword), but for stored procedures this is not compulsory.
- Stored procedures can use RETURN keyword but without any value being passed.
- Functions could be used in SELECT statements, provided they don,t do any data manipulation. However, procedures cannot be included in SELECT statements.
- A stored procedure can return multiple values using the OUT parameter or return no value at all.
- A stored procedure can save the query compilation time.

Comparison with prepared statements

Prepared statements take an ordinary statement or query and parameterize it so that different literal values can be used at a later time. Like stored procedures, they are stored on the server for efficiency and provide some protection from SQL injection attacks. Although simpler and more declarative, prepared statements are not ordinarily written to use procedural logic and cannot operate on variables. Because of their simple interface and client-side implementations, prepared statements are more widely reusable between DBMSs.

Disadvantages

- Stored procedure languages are quite often **vendor-specific**. Switching to another vendor's database most likely requires rewriting any existing stored procedures.
- Stored procedure languages from different vendors have different levels of sophistication.
  - For example, Oracle's PL/SQL has more language features and built-in features (via packages such as DBMS_ and UTL_ and others) than Microsoft's T-SQL.[*citation needed*]
- Tool support for writing and debugging stored procedures is often not as good as for other programming languages, but this differs between vendors and languages.
  - For example, both PL/SQL and T-SQL have dedicated IDEs and debuggers. PL/PgSQL can be debugged from various IDEs.

References

[1] Call Procedure (http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=/db2/rbafzmstcallstmt.htm)

External links

- Stored Procedures in MySQL FAQ (http://dev.mysql.com/doc/refman/5.7/en/faqs-stored-procs.html)
- An overview of PostgreSQL Procedural Language support (http://www.postgresql.org/docs/current/interactive/xplang.html)
- Using a stored procedure in Sybase ASE (http://www.petersap.nl/SybaseWiki/index.php/Stored_procedure)
- PL/SQL Procedures (http://infolab.stanford.edu/~ullman/fcdb/oracle/or-plsql.html#procedures)
- Oracle Database PL/SQL Language Reference (http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28370/toc.htm)

Cursor (databases)

In computer science, a database **cursor** is a control structure that enables traversal over the records in a database. Cursors facilitate subsequent processing in conjunction with the traversal, such as retrieval, addition and removal of database records. The database cursor characteristic of traversal makes cursors akin to the programming language concept of iterator.

Cursors are used by database programmers to process individual rows returned by database system queries. Cursors enable manipulation of whole result sets at once. In this scenario, a cursor enables the rows in a result set to be processed sequentially.

In SQL procedures, a cursor makes it possible to define a result set (a set of data rows) and perform complex logic on a row by row basis. By using the same mechanics, a SQL procedure can also define a result set and return it directly to the caller of the SQL procedure or to a client application.

A cursor can be viewed as a pointer to one row in a set of rows. The cursor can only reference one row at a time, but can move to other rows of the result set as needed.

Usage

To use cursors in SQL procedures, you need to do the following:

1. Declare a cursor that defines a result set.
2. Open the cursor to establish the result set.
3. Fetch the data into local variables as needed from the cursor, one row at a time.
4. Close the cursor when done.

To work with cursors you must use the following SQL statements

This section introduces the ways the SQL:2003 standard defines how to use cursors in applications in embedded SQL. Not all application bindings for relational database systems adhere to that standard, and some (such as CLI or JDBC) use a different interface.

A programmer makes a cursor known to the DBMS by using a DECLARE ... CURSOR statement and assigning the cursor a (compulsory) name:

```
DECLARE cursor_name CURSOR FOR SELECT ... FROM ...
```

Before code can access the data, it must open the cursor with the OPEN statement. Directly following a successful opening, the cursor is positioned *before* the first row in the result set.

```
OPEN cursor_name
```

Programs position cursors on a specific row in the result set with the FETCH statement. A fetch operation transfers the data of the row into the application.

```
FETCH cursor_name INTO ...
```

Once an application has processed all available rows or the fetch operation is to be positioned on a non-existing row (compare scrollable cursors below), the DBMS returns a SQLSTATE '02000' (usually accompanied by an SQLCODE +100) to indicate the end of the result set.

The final step involves closing the cursor using the CLOSE statement:

```
CLOSE cursor_name
```

After closing a cursor, a program can open it again, which implies that the DBMS re-evaluates the same query or a different query and builds a new result set.

Scrollable cursors

Programmers may declare cursors as scrollable or not scrollable. The scrollability indicates the direction in which a cursor can move.

With a **non-scrollable** (or **forward-only**) cursor, you can FETCH each row at most once, and the cursor automatically moves to the next row. After you fetch the last row, if you fetch again, you will put the cursor after the last row and get the following code: SQLSTATE 02000 (SQLCODE +100).

A program may position a **scrollable** cursor anywhere in the result set using the FETCH SQL statement. The keyword SCROLL must be specified when declaring the cursor. The default is NO SCROLL, although different language bindings like JDBC may apply a different default.

```
DECLARE cursor_name sensitivity SCROLL CURSOR FOR SELECT ... FROM ...
```

The target position for a scrollable cursor can be specified relatively (from the current cursor position) or absolutely (from the beginning of the result set).

```
FETCH [ NEXT | PRIOR | FIRST | LAST ] FROM cursor_name
```

```
FETCH ABSOLUTE n FROM cursor_name
```

```
FETCH RELATIVE n FROM cursor_name
```

Scrollable cursors can potentially access the same row in the result set multiple times. Thus, data modifications (insert, update, delete operations) from other transactions could have an impact on the result set. A cursor can be SENSITIVE or INSENSITIVE to such data modifications. A sensitive cursor picks up data modifications impacting the result set of the cursor, and an insensitive cursor does not. Additionally, a cursor may be ASENSITIVE, in which case the DBMS tries to apply sensitivity as much as possible.

"WITH HOLD"

Cursors are usually closed automatically at the end of a transaction, i.e. when a COMMIT or ROLLBACK (or an implicit termination of the transaction) occurs. That behavior can be changed if the cursor is declared using the WITH HOLD clause. (The default is WITHOUT HOLD.) A holdable cursor is kept open over COMMIT and closed upon ROLLBACK. (Some DBMS deviate from this standard behavior and also keep holdable cursors open over ROLLBACK.)

```
DECLARE cursor_name CURSOR WITH HOLD FOR SELECT ... FROM ...
```

When a COMMIT occurs, a holdable cursor is positioned *before* the next row. Thus, a positioned UPDATE or positioned DELETE statement will only succeed after a FETCH operation occurred first in the transaction.

Note that JDBC defines cursors as holdable per default. This is done because JDBC also activates auto-commit per default. Due to the usual overhead associated with auto-commit and holdable cursors, both features should be explicitly deactivated at the connection level.

Positioned update/delete statements

Cursors can not only be used to fetch data from the DBMS into an application but also to identify a row in a table to be updated or deleted. The SQL:2003 standard defines positioned update and positioned delete SQL statements for that purpose. Such statements do not use a regular WHERE clause with predicates. Instead, a cursor identifies the row. The cursor must be opened and already positioned on a row by means of FETCH statement.

```
UPDATE table_name
SET    ...
WHERE  CURRENT OF cursor_name
```

```
DELETE
FROM   table_name
WHERE  CURRENT OF cursor_name
```

The cursor must operate on an updatable result set in order to successfully execute a positioned update or delete statement. Otherwise, the DBMS would not know how to apply the data changes to the underlying tables referred to in the cursor.

Cursors in distributed transactions

Using cursors in distributed transactions (X/Open XA Environments), which are controlled using a transaction monitor, is no different than cursors in non-distributed transactions.

One has to pay attention when using holdable cursors, however. Connections can be used by different applications. Thus, once a transaction has been ended and committed, a subsequent transaction (running in a different application) could inherit existing holdable cursors. Therefore, an application developer has to be aware of that situation.

Cursors in XQuery

The XQuery language allows cursors to be created using the **subsequence()** function.

The format is:

```
let $displayed-sequence := subsequence($result, $start, $item-count)
```

Where $**result** is the result of the initial XQuery, **$start** is the item number to start and **$item-count** is the number of items to return.

Equivalently this can also be done using a predicate:

```
let $displayed-sequence := $result[$start to $end]
```

Where $end is the end sequence.

For complete examples see the XQuery Wikibook [1].

Disadvantages of cursors

The following information may vary depending on the specific database system.

Fetching a row from the cursor may result in a network round trip each time. This uses much more network bandwidth than would ordinarily be needed for the execution of a single SQL statement like DELETE. Repeated network round trips can severely impact the speed of the operation using the cursor. Some DBMSs try to reduce this impact by using block fetch. Block fetch implies that multiple rows are sent together from the server to the client. The client stores a whole block of rows in a local buffer and retrieves the rows from there until that buffer is exhausted.

Cursors allocate resources on the server, for instance locks, packages, processes, temporary storage, etc. For example, Microsoft SQL Server implements cursors by creating a temporary table and populating it with the query's result set. If a cursor is not properly closed (*deallocated*), the resources will not be freed until the SQL session (connection) itself is closed. This wasting of resources on the server can not only lead to performance degradations but also to failures.

Example

```
EMPLOYEES TABLE
SQL> desc EMPLOYEES_DETAILS;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------

 EMPLOYEE_ID                               NOT NULL NUMBER(6)
 FIRST_NAME                                         VARCHAR2(20)
 LAST_NAME                                 NOT NULL VARCHAR2(25)
 EMAIL                                     NOT NULL VARCHAR2(30)
 PHONE_NUMBER                                       VARCHAR2(20)
 HIRE_DATE                                 NOT NULL DATE
 JOB_ID                                    NOT NULL VARCHAR2(10)
 SALARY                                             NUMBER(8,2)
 COMMISSION_PCT                                     NUMBER(2,2)
 MANAGER_ID                                         NUMBER(6)
 DEPARTMENT_ID                                      NUMBER(4)

SAMPLE CURSOR KNOWN AS EE

CREATE OR REPLACE
PROCEDURE EE AS
BEGIN
      DECLARE
v_employeeID EMPLOYEES_DETAILS.EMPLOYEE_ID%TYPE;
v_FirstName EMPLOYEES_DETAILS.FIRST_NAME%TYPE;
v_LASTName EMPLOYEES_DETAILS.LAST_NAME%TYPE;

v_JOB_ID EMPLOYEES_DETAILS.JOB_ID%TYPE:= 'IT_PROG';

Cursor c_EMPLOYEES_DETAILS IS
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME
```

```
FROM EMPLOYEES_DETAILS
WHERE JOB_ID ='v_JOB_ID';
BEGIN
OPEN c_EMPLOYEES_DETAILS;

LOOP

FETCH c_EMPLOYEES_DETAILS INTO v_employeeID,v_FirstName,v_LASTName;

DBMS_OUTPUT.put_line( v_employeeID);
DBMS_OUTPUT.put_line( v_FirstName);
DBMS_OUTPUT.put_line( v_LASTName);

EXIT WHEN c_EMPLOYEES_DETAILS%NOTFOUND;
END LOOP;

CLOSE c_EMPLOYEES_DETAILS;
END;

END;
```

References

- Christopher J. Date: *Database in Depth*, O'Reilly & Associates, ISBN 0-596-10012-4
- Thomas M. Connolly, Carolyn E. Begg: *Database Systems*, Addison-Wesley, ISBN 0-321-21025-5
- Ramiz Elmasri, Shamkant B. Navathe: *Fundamentals of Database Systems*, Addison-Wesley, ISBN 0-201-54263-3
- Neil Matthew, Richard Stones: *Beginning Databases with PostgreSQL: From Novice to Professional*, Apress, ISBN 1-59059-478-9
- Thomas Kyte: *Expert One-On-One: Oracle*, Apress, ISBN 1-59059-525-4
- Kevin Loney: *Oracle Database 10g: The Complete Reference*, Oracle Press, ISBN 0-07-225351-7

External links

- Cursor Optimization Tips (for MS SQL Server) [2]
- Descriptions from Portland Pattern Repository [3]
- PostgreSQL Documentation [4]
- Berkeley DB Reference Guide: Cursor operations [5]
- Java SE 7 [6]
- Q3SqlCursor Class Reference [7]
- OCI Scrollable Cursor [8]
- function oci_new_cursor [9]
- MySQL's Cursor Documentation [10]
- FirebirdSQL cursors documentation [11]
- Cursors in DB2 CLI applications [12]; Cursors in DB2 SQL stored procedures [13]
- A Simple Example of a MySQL Stored Procedure that uses a cursor [14]
- MariaDB/MySQL Cursors: a brief Tutorial [15]

References

[1] http://en.wikibooks.org/wiki/XQuery/Searching,Paging_and_Sorting#Paging
[2] http://www.mssqlcity.com/Tips/tipCursor.htm
[3] http://c2.com/cgi/wiki?DistributedCursor
[4] http://www.postgresql.org/docs/8.3/interactive/plpgsql-cursors.html
[5] http://sleepycat.com/docs/ref/am/cursor.html
[6] http://download.oracle.com/javase/7/docs/
[7] http://doc.trolltech.com/4.0/q3sqlcursor.html
[8] http://www.oracle.com/technology/products/oracle9i/daily/mar15.html
[9] http://de2.php.net/manual/en/function.oci-new-cursor.php
[10] http://dev.mysql.com/doc/refman/5.5/en/cursors.html
[11] http://www.firebirdsql.org/refdocs/langrefupd20-psql-declare.html
[12] http://publib.boulder.ibm.com/infocenter/db2luw/v9/topic/com.ibm.db2.udb.apdv.cli.doc/doc/c0007645.htm
[13] http://publib.boulder.ibm.com/infocenter/db2luw/v9/topic/com.ibm.db2.udb.apdv.sql.doc/doc/c0024361.htm
[14] http://www.kbedell.com/2009/03/02/a-simple-example-of-a-mysql-stored-procedure-that-uses-a-cursor/
[15] http://falseisnotnull.wordpress.com/2013/06/05/mariadbmysql-cursors-a-brief-tutorial/

Partition (database)

A **partition** is a division of a logical database or its constituting elements into distinct independent parts. Database partitioning is normally done for manageability, performance or availability reasons.

Benefits of multiple partitions

A popular and favourable application of partitioning is in a distributed database management system. Each partition may be spread over multiple nodes, and users at the node can perform local transactions on the partition. This increases performance for sites that have regular transactions involving certain views of data, whilst maintaining availability and security.

Partitioning criteria

Current high end relational database management systems provide for different criteria to split the database. They take a *partitioning key* and assign a partition based on certain criteria. Common criteria are:

Range partitioning

> Selects a partition by determining if the partitioning key is inside a certain range. An example could be a partition for all rows where the column `zipcode` has a value between `70000` and `79999`.

List partitioning

> A partition is assigned a list of values. If the partitioning key has one of these values, the partition is chosen. For example all rows where the column `Country` is either `Iceland`, `Norway`, `Sweden`, `Finland` or `Denmark` could build a partition for the Nordic countries.

Hash partitioning

> The value of a hash function determines membership in a partition. Assuming there are four partitions, the hash function could return a value from 0 to 3.

Composite partitioning allows for certain combinations of the above partitioning schemes, by for example first applying a range partitioning and then a hash partitioning. Consistent hashing could be considered a composite of hash and list partitioning where the hash reduces the key space to a size that can be listed.

Partitioning methods

The partitioning can be done by either building separate smaller databases (each with its own tables, indices, and transaction logs), or by splitting selected elements, for example just one table.

Horizontal partitioning (also see ***shard***) involves putting different rows into different tables. Perhaps customers with ZIP codes less than 50000 are stored in CustomersEast, while customers with ZIP codes greater than or equal to 50000 are stored in CustomersWest. The two partition tables are then CustomersEast and CustomersWest, while a view with a union might be created over both of them to provide a complete view of all customers.

Vertical partitioning involves creating tables with fewer columns and using additional tables to store the remaining columns.[1] Normalization also involves this splitting of columns across tables, but vertical partitioning goes beyond that and partitions columns even when already normalized. Different physical storage might be used to realize vertical partitioning as well; storing infrequently used or very wide columns on a different device, for example, is a method of vertical partitioning. Done explicitly or implicitly, this type of partitioning is called "row splitting" (the row is split by its columns). A common form of vertical partitioning is to split dynamic data (slow to find) from static data (fast to find) in a table where the dynamic data is not used as often as the static. Creating a view across the two newly created tables restores the original table with a performance penalty, however performance will increase when accessing the static data e.g. for statistical analysis.

References

[1] Vertical Partitioning Algorithms for Database Design, by Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou, Stanford University 1984 (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.97.8306)

External links

- IBM DB2 partitioning (http://publib.boulder.ibm.com/infocenter/db2help/index.jsp?topic=/com.ibm.db2.udb.doc/admin/c0004885.htm)
- MySQL partitioning (http://dev.mysql.com/doc/refman/5.5/en/partitioning.html)
- Oracle partitioning (http://www.oracle.com/us/products/database/options/partitioning/index.htm)
- SQL Server partitions (http://msdn.microsoft.com/en-us/library/ms190787.aspx)
- PostgreSQL partitioning (http://www.postgresql.org/docs/current/interactive/ddl-partitioning.html)
- Sybase ASE 15.0 partitioning (http://www.sybase.com/detail?id=1036923)
- MongoDB partitioning (http://www.mongodb.org/display/DOCS/Sharding)
- ScimoreDB partitioning (http://scimore.com/wiki/Distributed_schema)
- VoltDB partitioning (http://community.voltdb.com/docs/UsingVoltDB/ChapAppDesign#DesignPartition)

Components

Concurrency control

In information technology and computer science, especially in the fields of computer programming, operating systems, multiprocessors, and databases, **concurrency control** ensures that correct results for concurrent operations are generated, while getting those results as quickly as possible.

Computer systems, both software and hardware, consist of modules, or components. Each component is designed to operate correctly, i.e., to obey or to meet certain consistency rules. When components that operate concurrently interact by messaging or by sharing accessed data (in memory or storage), a certain component's consistency may be violated by another component. The general area of concurrency control provides rules, methods, design methodologies, and theories to maintain the consistency of components operating concurrently while interacting, and thus the consistency and correctness of the whole system. Introducing concurrency control into a system means applying operation constraints which typically result in some performance reduction. Operation consistency and correctness should be achieved with as good as possible efficiency, without reducing performance below reasonable.

For example, a failure in concurrency control can result in data corruption from torn read or write operations.

Concurrency control in databases

Comments:

1. This section is applicable to all transactional systems, i.e., to all systems that use *database transactions* (*atomic transactions*; e.g., transactional objects in Systems management and in networks of smartphones which typically implement private, dedicated database systems), not only general-purpose database management systems (DBMSs).
2. DBMSs need to deal also with concurrency control issues not typical just to database transactions but rather to operating systems in general. These issues (e.g., see *Concurrency control in operating systems* below) are out of the scope of this section.

Concurrency control in Database management systems (DBMS; e.g., Bernstein et al. 1987, Weikum and Vossen 2001), other transactional objects, and related distributed applications (e.g., Grid computing and Cloud computing) ensures that *database transactions* are performed concurrently without violating the data integrity of the respective databases. Thus concurrency control is an essential element for correctness in any system where two database transactions or more, executed with time overlap, can access the same data, e.g., virtually in any general-purpose database system. Consequently a vast body of related research has been accumulated since database systems emerged in the early 1970s. A well established concurrency control theory for database systems is outlined in the references mentioned above: serializability theory, which allows to effectively design and analyze concurrency control methods and mechanisms. An alternative theory for concurrency control of atomic transactions over abstract data types is presented in (Lynch et al. 1993), and not utilized below. This theory is more refined, complex, with a wider scope, and has been less utilized in the Database literature than the classical theory above. Each theory has its pros and cons, emphasis and insight. To some extent they are complementary, and their merging may be useful.

To ensure correctness, a DBMS usually guarantees that only *serializable* transaction schedules are generated, unless *serializability* is intentionally relaxed to increase performance, but only in cases where application correctness is not harmed. For maintaining correctness in cases of failed (aborted) transactions (which can always happen for many reasons) schedules also need to have the *recoverability* (from abort) property. A DBMS also guarantees that no effect of *committed* transactions is lost, and no effect of *aborted* (rolled back) transactions remains in the related

database. Overall transaction characterization is usually summarized by the ACID rules below. As databases have become distributed, or needed to cooperate in distributed environments (e.g., Federated databases in the early 1990, and Cloud computing currently), the effective distribution of concurrency control mechanisms has received special attention.

Database transaction and the ACID rules

The concept of a *database transaction* (or *atomic transaction*) has evolved in order to enable both a well understood database system behavior in a faulty environment where crashes can happen any time, and *recovery* from a crash to a well understood database state. A database transaction is a unit of work, typically encapsulating a number of operations over a database (e.g., reading a database object, writing, acquiring lock, etc.), an abstraction supported in database and also other systems. Each transaction has well defined boundaries in terms of which program/code executions are included in that transaction (determined by the transaction's programmer via special transaction commands). Every database transaction obeys the following rules (by support in the database system; i.e., a database system is designed to guarantee them for the transactions it runs):

- **Atomicity** - Either the effects of all or none of its operations remain ("all or nothing" semantics) when a transaction is completed (*committed* or *aborted* respectively). In other words, to the outside world a committed transaction appears (by its effects on the database) to be indivisible, atomic, and an aborted transaction does not leave effects on the database at all, as if never existed.
- **Consistency** - Every transaction must leave the database in a consistent (correct) state, i.e., maintain the predetermined integrity rules of the database (constraints upon and among the database's objects). A transaction must transform a database from one consistent state to another consistent state (however, it is the responsibility of the transaction's programmer to make sure that the transaction itself is correct, i.e., performs correctly what it intends to perform (from the application's point of view) while the predefined integrity rules are enforced by the DBMS). Thus since a database can be normally changed only by transactions, all the database's states are consistent. An aborted transaction does not change the database state it has started from, as if it never existed (atomicity above).
- **Isolation** - Transactions cannot interfere with each other (as an end result of their executions). Moreover, usually (depending on concurrency control method) the effects of an incomplete transaction are not even visible to another transaction. Providing isolation is the main goal of concurrency control.
- **Durability** - Effects of successful (committed) transactions must persist through crashes (typically by recording the transaction's effects and its commit event in a non-volatile memory).

The concept of atomic transaction has been extended during the years to what has become Business transactions which actually implement types of Workflow and are not atomic. However also such enhanced transactions typically utilize atomic transactions as components.

Why is concurrency control needed?

If transactions are executed *serially*, i.e., sequentially with no overlap in time, no transaction concurrency exists. However, if concurrent transactions with interleaving operations are allowed in an uncontrolled manner, some unexpected, undesirable result may occur. Here are some typical examples:

1. The lost update problem: A second transaction writes a second value of a data-item (datum) on top of a first value written by a first concurrent transaction, and the first value is lost to other transactions running concurrently which need, by their precedence, to read the first value. The transactions that have read the wrong value end with incorrect results.
2. The dirty read problem: Transactions read a value written by a transaction that has been later aborted. This value disappears from the database upon abort, and should not have been read by any transaction ("dirty read"). The reading transactions end with incorrect results.

3. The incorrect summary problem: While one transaction takes a summary over the values of all the instances of a repeated data-item, a second transaction updates some instances of that data-item. The resulting summary does not reflect a correct result for any (usually needed for correctness) precedence order between the two transactions (if one is executed before the other), but rather some random result, depending on the timing of the updates, and whether certain update results have been included in the summary or not.

Most high-performance transactional systems need to run transactions concurrently to meet their performance requirements. Thus, without concurrency control such systems can neither provide correct results nor maintain their databases consistent.

Concurrency control mechanisms

Categories

The main categories of concurrency control mechanisms are:

- **Optimistic** - Delay the checking of whether a transaction meets the isolation and other integrity rules (e.g., serializability and recoverability) until its end, without blocking any of its (read, write) operations ("...and be optimistic about the rules being met..."), and then abort a transaction to prevent the violation, if the desired rules are to be violated upon its commit. An aborted transaction is immediately restarted and re-executed, which incurs an obvious overhead (versus executing it to the end only once). If not too many transactions are aborted, then being optimistic is usually a good strategy.
- **Pessimistic** - Block an operation of a transaction, if it may cause violation of the rules, until the possibility of violation disappears. Blocking operations is typically involved with performance reduction.
- **Semi-optimistic** - Block operations in some situations, if they may cause violation of some rules, and do not block in other situations while delaying rules checking (if needed) to transaction's end, as done with optimistic.

Different categories provide different performance, i.e., different average transaction completion rates (*throughput*), depending on transaction types mix, computing level of parallelism, and other factors. If selection and knowledge about trade-offs are available, then category and method should be chosen to provide the highest performance.

The mutual blocking between two transactions (where each one blocks the other) or more results in a deadlock, where the transactions involved are stalled and cannot reach completion. Most non-optimistic mechanisms (with blocking) are prone to deadlocks which are resolved by an intentional abort of a stalled transaction (which releases the other transactions in that deadlock), and its immediate restart and re-execution. The likelihood of a deadlock is typically low.

Both blocking, deadlocks, and aborts result in performance reduction, and hence the trade-offs between the categories.

Methods

Many methods for concurrency control exist. Most of them can be implemented within either main category above. The major methods,[1] which have each many variants, and in some cases may overlap or be combined, are:

1. Locking (e.g., **Two-phase locking** - 2PL) - Controlling access to data by locks assigned to the data. Access of a transaction to a data item (database object) locked by another transaction may be blocked (depending on lock type and access operation type) until lock release.
2. **Serialization graph checking** (also called Serializability, or Conflict, or Precedence graph checking) - Checking for cycles in the schedule's graph and breaking them by aborts.
3. **Timestamp ordering** (TO) - Assigning timestamps to transactions, and controlling or checking access to data by timestamp order.
4. **Commitment ordering** (or Commit ordering; CO) - Controlling or checking transactions' chronological order of commit events to be compatible with their respective precedence order.

Other major concurrency control types that are utilized in conjunction with the methods above include:

- **Multiversion concurrency control** (MVCC) - Increasing concurrency and performance by generating a new version of a database object each time the object is written, and allowing transactions' read operations of several last relevant versions (of each object) depending on scheduling method.
- **Index concurrency control** - Synchronizing access operations to indexes, rather than to user data. Specialized methods provide substantial performance gains.
- **Private workspace model** (**Deferred update**) - Each transaction maintains a private workspace for its accessed data, and its changed data become visible outside the transaction only upon its commit (e.g., Weikum and Vossen 2001). This model provides a different concurrency control behavior with benefits in many cases.

The most common mechanism type in database systems since their early days in the 1970s has been *Strong strict Two-phase locking* (SS2PL; also called *Rigorous scheduling* or *Rigorous 2PL*) which is a special case (variant) of both Two-phase locking (2PL) and Commitment ordering (CO). It is pessimistic. In spite of its long name (for historical reasons) the idea of the **SS2PL** mechanism is simple: "Release all locks applied by a transaction only after the transaction has ended." SS2PL (or Rigorousness) is also the name of the set of all schedules that can be generated by this mechanism, i.e., these are SS2PL (or Rigorous) schedules, have the SS2PL (or Rigorousness) property.

Major goals of concurrency control mechanisms

Concurrency control mechanisms firstly need to operate correctly, i.e., to maintain each transaction's integrity rules (as related to concurrency; application-specific integrity rule are out of the scope here) while transactions are running concurrently, and thus the integrity of the entire transactional system. Correctness needs to be achieved with as good performance as possible. In addition, increasingly a need exists to operate effectively while transactions are distributed over processes, computers, and computer networks. Other subjects that may affect concurrency control are recovery and replication.

Correctness

Serializability

For correctness, a common major goal of most concurrency control mechanisms is generating schedules with the *Serializability* property. Without serializability undesirable phenomena may occur, e.g., money may disappear from accounts, or be generated from nowhere. **Serializability** of a schedule means equivalence (in the resulting database values) to some *serial* schedule with the same transactions (i.e., in which transactions are sequential with no overlap in time, and thus completely isolated from each other: No concurrent access by any two transactions to the same data is possible). Serializability is considered the highest level of isolation among database transactions, and the major correctness criterion for concurrent transactions. In some cases compromised, relaxed forms of serializability are allowed for better performance (e.g., the popular *Snapshot isolation* mechanism) or to meet availability requirements in highly distributed systems (see *Eventual consistency*), but only if application's correctness is not violated by the relaxation (e.g., no relaxation is allowed for money transactions, since by relaxation money can disappear, or appear from nowhere).

Almost all implemented concurrency control mechanisms achieve serializability by providing *Conflict serializablity*, a broad special case of serializability (i.e., it covers, enables most serializable schedules, and does not impose significant additional delay-causing constraints) which can be implemented efficiently.

Recoverability

See *Recoverability* in *Serializability*

Comment: While in the general area of systems the term "recoverability" may refer to the ability of a system to recover from failure or from an incorrect/forbidden state, within concurrency control of database systems this term has received a specific meaning.

Concurrency control typically also ensures the *Recoverability* property of schedules for maintaining correctness in cases of aborted transactions (which can always happen for many reasons). **Recoverability** (from abort) means that no committed transaction in a schedule has read data written by an aborted transaction. Such data disappear from the database (upon the abort) and are parts of an incorrect database state. Reading such data violates the consistency rule of ACID. Unlike Serializability, Recoverability cannot be compromised, relaxed at any case, since any relaxation results in quick database integrity violation upon aborts. The major methods listed above provide serializability mechanisms. None of them in its general form automatically provides recoverability, and special considerations and mechanism enhancements are needed to support recoverability. A commonly utilized special case of recoverability is *Strictness*, which allows efficient database recovery from failure (but excludes optimistic implementations; e.g., Strict CO (SCO) cannot have an optimistic implementation, but has semi-optimistic ones).

Comment: Note that the *Recoverability* property is needed even if no database failure occurs and no database *recovery* from failure is needed. It is rather needed to correctly automatically handle transaction aborts, which may be unrelated to database failure and recovery from it.

Distribution

With the fast technological development of computing the difference between local and distributed computing over low latency networks or buses is blurring. Thus the quite effective utilization of local techniques in such distributed environments is common, e.g., in computer clusters and multi-core processors. However the local techniques have their limitations and use multi-processes (or threads) supported by multi-processors (or multi-cores) to scale. This often turns transactions into distributed ones, if they themselves need to span multi-processes. In these cases most local concurrency control techniques do not scale well.

Distributed serializability and Commitment ordering

See *Distributed serializability* in *Serializability*

As database systems have become distributed, or started to cooperate in distributed environments (e.g., Federated databases in the early 1990s, and nowadays Grid computing, Cloud computing, and networks with smartphones), some transactions have become distributed. A distributed transaction means that the transaction spans processes, and may span computers and geographical sites. This generates a need in effective distributed concurrency control mechanisms. Achieving the Serializability property of a distributed system's schedule (see *Distributed serializability* and *Global serializability* (*Modular serializability*)) effectively poses special challenges typically not met by most of the regular serializability mechanisms, originally designed to operate locally. This is especially due to a need in costly distribution of concurrency control information amid communication and computer latency. The only known general effective technique for distribution is Commitment ordering, which was disclosed publicly in 1991 (after being patented). **Commitment ordering** (Commit ordering, CO; Raz 1992) means that transactions' chronological order of commit events is kept compatible with their respective precedence order. CO does not require the distribution of concurrency control information and provides a general effective solution (reliable, high-performance, and scalable) for both distributed and global serializability, also in a heterogeneous environment with database systems (or other transactional objects) with different (any) concurrency control mechanisms. CO is indifferent to which mechanism is utilized, since it does not interfere with any transaction operation scheduling (which most mechanisms control), and only determines the order of commit events. Thus, CO enables the efficient distribution of all other mechanisms, and also the distribution of a mix of different (any) local mechanisms, for achieving

distributed and global serializability. The existence of such a solution has been considered "unlikely" until 1991, and by many experts also later, due to misunderstanding of the CO solution (see Quotations in *Global serializability*). An important side-benefit of CO is automatic distributed deadlock resolution. Contrary to CO, virtually all other techniques (when not combined with CO) are prone to distributed deadlocks (also called global deadlocks) which need special handling. CO is also the name of the resulting schedule property: A schedule has the CO property if the chronological order of its transactions' commit events is compatible with the respective transactions' precedence (partial) order.

SS2PL mentioned above is a variant (special case) of CO and thus also effective to achieve distributed and global serializability. It also provides automatic distributed deadlock resolution (a fact overlooked in the research literature even after CO's publication), as well as Strictness and thus Recoverability. Possessing these desired properties together with known efficient locking based implementations explains SS2PL's popularity. SS2PL has been utilized to efficiently achieve Distributed and Global serializability since the 1980, and has become the de facto standard for it. However, SS2PL is blocking and constraining (pessimistic), and with the proliferation of distribution and utilization of systems different from traditional database systems (e.g., as in Cloud computing), less constraining types of CO (e.g., Optimistic CO) may be needed for better performance.

Comments:

1. The *Distributed conflict serializability* property in its general form is difficult to achieve efficiently, but it is achieved efficiently via its special case *Distributed CO*: Each local component (e.g., a local DBMS) needs both to provide some form of CO, and enforce a special *vote ordering strategy* for the *Two-phase commit protocol* (2PC: utilized to commit distributed transactions). Differently from the general Distributed CO, *Distributed SS2PL* exists automatically when all local components are SS2PL based (in each component CO exists, implied, and the vote ordering strategy is now met automatically). This fact has been known and utilized since the 1980s (i.e., that SS2PL exists globally, without knowing about CO) for efficient Distributed SS2PL, which implies Distributed serializability and strictness (e.g., see Raz 1992, page 293; it is also implied in Bernstein et al. 1987, page 78). Less constrained Distributed serializability and strictness can be efficiently achieved by Distributed Strict CO (SCO), or by a mix of SS2PL based and SCO based local components.
2. About the references and Commitment ordering: (Bernstein et al. 1987) was published before the discovery of CO in 1990. The CO schedule property is called *Dynamic atomicity* in (Lynch et al. 1993, page 201). CO is described in (Weikum and Vossen 2001, pages 102, 700), but the description is partial and misses CO's essence. (Raz 1992) was the first refereed and accepted for publication article about CO algorithms (however, publications about an equivalent Dynamic atomicity property can be traced to 1988). Other CO articles followed. (Bernstein and Newcomer 2009) note CO as one of the four major concurrency control methods, and CO's ability to provide interoperability among other methods.

Distributed recoverability

Unlike Serializability, *Distributed recoverability* and *Distributed strictness* can be achieved efficiently in a straightforward way, similarly to the way Distributed CO is achieved: In each database system they have to be applied locally, and employ a vote ordering strategy for the Two-phase commit protocol (2PC; Raz 1992, page 307).

As has been mentioned above, Distributed SS2PL, including Distributed strictness (recoverability) and Distributed commitment ordering (serializability), automatically employs the needed vote ordering strategy, and is achieved (globally) when employed locally in each (local) database system (as has been known and utilized for many years; as a matter of fact locality is defined by the boundary of a 2PC participant (Raz 1992)).

Other major subjects of attention

The design of concurrency control mechanisms is often influenced by the following subjects:

Recovery

All systems are prone to failures, and handling *recovery* from failure is a must. The properties of the generated schedules, which are dictated by the concurrency control mechanism, may have an impact on the effectiveness and efficiency of recovery. For example, the Strictness property (mentioned in the section Recoverability above) is often desirable for an efficient recovery.

Replication

For high availability database objects are often *replicated*. Updates of replicas of a same database object need to be kept synchronized. This may affect the way concurrency control is done (e.g., Gray et al. 1996).

References

- Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman (1987): *Concurrency Control and Recovery in Database Systems* [2] (free PDF download), Addison Wesley Publishing Company, 1987, ISBN 0-201-10715-5
- Gerhard Weikum, Gottfried Vossen (2001): *Transactional Information Systems* [3], Elsevier, ISBN 1-55860-508-8
- Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete (1993): *Atomic Transactions in Concurrent and Distributed Systems* [4], Morgan Kauffman (Elsevier), August 1993, ISBN 978-1-55860-104-8, ISBN 1-55860-104-X
- Yoav Raz (1992): "The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment." [5] (PDF [6]), *Proceedings of the Eighteenth International Conference on Very Large Data Bases* (VLDB), pp. 292-312, Vancouver, Canada, August 1992. (also DEC-TR 841, Digital Equipment Corporation, November 1990)

Footnotes

[1] Philip A. Bernstein, Eric Newcomer (2009): *Principles of Transaction Processing*, 2nd Edition (http://www.elsevierdirect.com/product.jsp?isbn=9781558606234), Morgan Kaufmann (Elsevier), June 2009, ISBN 978-1-55860-623-4 (page 145)
[2] http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx
[3] http://www.elsevier.com/wps/find/bookdescription.cws_home/677937/description#description
[4] http://www.elsevier.com/wps/find/bookdescription.cws_home/680521/description#description
[5] http://www.informatik.uni-trier.de/~ley/db/conf/vldb/Raz92.html
[6] http://www.vldb.org/conf/1992/P292.PDF

Concurrency control in operating systems

Multitasking operating systems, especially real-time operating systems, need to maintain the illusion that all tasks running on top of them are all running at the same time, even though only one or a few tasks really are running at any given moment due to the limitations of the hardware the operating system is running on. Such multitasking is fairly simple when all tasks are independent from each other. However, when several tasks try to use the same resource, or when tasks try to share information, it can lead to confusion and inconsistency. The task of concurrent computing is to solve that problem. Some solutions involve "locks" similar to the locks used in databases, but they risk causing problems of their own such as deadlock. Other solutions are Non-blocking algorithms.

References

- Andrew S. Tanenbaum, Albert S Woodhull (2006): *Operating Systems Design and Implementation, 3rd Edition*, Prentice Hall, ISBN 0-13-142938-8
- Silberschatz, Avi; Galvin, Peter; Gagne, Greg (2008). *Operating Systems Concepts, 8th edition*. John Wiley & Sons. ISBN 0-470-12872-0.

Data dictionary

A **data dictionary**, or metadata repository, as defined in the *IBM Dictionary of Computing*, is a "centralized repository of information about data such as meaning, relationships to other data, origin, usage, and format."[1] The term may have one of several closely related meanings pertaining to databases and database management systems (DBMS):

- a document describing a database or collection of databases
- an integral component of a DBMS that is required to determine its structure
- a piece of middleware that extends or supplants the native data dictionary of a DBMS

Documentation

The term **data dictionary** and **data repository** are used to indicate a more general software utility than a catalogue. A **catalogue** is closely coupled with the DBMS software. It provides the information stored in it to the user and the DBA, but it is mainly accessed by the various software modules of the DBMS itself, such as DDL and DML compilers, the query optimiser, the transaction processor, report generators, and the constraint enforcer. On the other hand, a **data dictionary** is a data structure that stores metadata, i.e., (structured) data about data. The software package for a stand-alone data dictionary or data repository may interact with the software modules of the DBMS, but it is mainly used by the designers, users and administrators of a computer system for information resource management. These systems are used to maintain information on system hardware and software configuration, documentation, application and users as well as other information relevant to system administration.[2]

If a data dictionary system is used only by the designers, users, and administrators and not by the DBMS Software, it is called a **passive data dictionary.** Otherwise, it is called an **active data dictionary** or **data dictionary.** When a passive data dictionary is updated, it is done so manually and independently from any changes to a DBMS (database) structure. With an active data dictionary, the dictionary is updated first and changes occur in the DBMS automatically as a result.

Database users and application developers can benefit from an authoritative data dictionary document that catalogs the organization, contents, and conventions of one or more databases.[3] This typically includes the names and descriptions of various tables (records or Entities) and their contents (fields) plus additional details, like the type and length of each data element. Another important piece of information that a data dictionary can provide is the relationship between Tables. This is sometimes referred to in Entity-Relationship diagrams, or if using Set descriptors, identifying in which Sets database Tables participate.

In an active data dictionary constraints may be placed upon the underlying data. For instance, a Range may be imposed on the value of numeric data in a data element (field), or a Record in a Table may be FORCED to participate in a set relationship with another Record-Type. Additionally, a distributed DBMS may have certain location specifics described within its active data dictionary (e.g. where Tables are physically located).

The data dictionary consists of record types (tables) created in the database by systems generated command files, tailored for each supported back-end DBMS. Command files contain SQL Statements for CREATE TABLE, CREATE UNIQUE INDEX, ALTER TABLE (for referential integrity), etc., using the specific statement required by

that type of database.

There is no universal standard as to the level of detail in such a document.

Middleware

In the construction of database applications, it can be useful to introduce an additional layer of data dictionary software, i.e. middleware, which communicates with the underlying DBMS data dictionary. Such a "high-level" data dictionary may offer additional features and a degree of flexibility that goes beyond the limitations of the native "low-level" data dictionary, whose primary purpose is to support the basic functions of the DBMS, not the requirements of a typical application. For example, a high-level data dictionary can provide alternative entity-relationship models tailored to suit different applications that share a common database.[4] Extensions to the data dictionary also can assist in query optimization against distributed databases.[5] Additionally, DBA functions are often automated using restructuring tools that are tightly coupled to an active data dictionary.

Software frameworks aimed at rapid application development sometimes include high-level data dictionary facilities, which can substantially reduce the amount of programming required to build menus, forms, reports, and other components of a database application, including the database itself. For example, PHPLens includes a PHP class library to automate the creation of tables, indexes, and foreign key constraints portably for multiple databases.[6] Another PHP-based data dictionary, part of the RADICORE toolkit, automatically generates program objects, scripts, and SQL code for menus and forms with data validation and complex joins.[7] For the ASP.NET environment, Base One's data dictionary provides cross-DBMS facilities for automated database creation, data validation, performance enhancement (caching and index utilization), application security, and extended data types.[8] Visual DataFlex features[9] provides the ability to use DataDictionaries as class files to form middle layer between the user interface and the underlying database. The intent is to create standardized rules to maintain data integrity and enforce business rules throughout one or more related applications.

References

[1] ACM, IBM Dictionary of Computing (http://portal.acm.org/citation.cfm?id=541721), 10th edition, 1993
[2] Ramez Elmasri, Shamkant B. Navathe: *Fundamentals of Database Systems*, 3rd. ed. sect. 17.5, p. 582
[3] TechTarget, *SearchSOA*, What is a data dictionary? (http://searchsoa.techtarget.com/sDefinition/0,,sid26_gci211896,00.html)
[4] U.S. Patent 4774661, Database management system with active data dictionary (http://www.freepatentsonline.com/4774661.html), 19 November 1985, AT&T
[5] U.S. Patent 4769772, Automated query optimization method using both global and parallel local optimizations for materialization access planning for distributed databases (http://www.freepatentsonline.com/4769772.html), 28 February 1985, Honeywell Bull
[6] PHPLens, ADOdb Data Dictionary Library for PHP (http://phplens.com/lens/adodb/docs-datadict.htm)
[7] RADICORE, What is a Data Dictionary? (http://www.radicore.org/viewarticle.php?article_id=5)
[8] Base One International Corp., Base One Data Dictionary (http://www.boic.com/b1ddic.htm)
[9] VISUAL DATAFLEX, features (http://www.visualdataflex.com/features.asp?pageid=1030)

External links

- Yourdon, *Structured Analysis Wiki*, Data Dictionaries (http://yourdon.com/strucanalysis/wiki/index.php?title=Chapter_10)

Java Database Connectivity

JDBC

Type	Data Access API
Website	Java SE 7 [6]

JDBC is a Java-based data access technology (Java Standard Edition platform) from Oracle Corporation. This technology is an API for the Java programming language that defines how a client may access a database. It provides methods for querying and updating data in a database. JDBC is oriented towards relational databases. A JDBC-to-ODBC bridge enables connections to any ODBC-accessible data source in the JVM host environment.

History and implementation

Sun Microsystems released JDBC as part of JDK 1.1 on February 19, 1997. It has since formed part of the Java Standard Edition.

The JDBC classes are contained in the Java package `java.sql` [1] and `javax.sql` [2].

Starting with version 3.1, JDBC has been developed under the Java Community Process. JSR 54 specifies JDBC 3.0 (included in J2SE 1.4), JSR 114 specifies the JDBC Rowset additions, and JSR 221 is the specification of JDBC 4.0 (included in Java SE 6).[3]

The latest version, JDBC 4.1, is specified by a maintenance release of JSR 221[4] and is included in Java SE 7.[5]

Functionality

JDBC allows multiple implementations to exist and be used by the same application. The API provides a mechanism for dynamically loading the correct Java packages and registering them with the JDBC Driver Manager. The Driver Manager is used as a connection factory for creating JDBC connections.

JDBC connections support creating and executing statements. These may be update statements such as SQL's CREATE, INSERT, UPDATE and DELETE, or they may be query statements such as SELECT. Additionally, stored procedures may be invoked through a JDBC connection. JDBC represents statements using one of the following classes:

- `Statement` [6] – the statement is sent to the database server each and every time.
- `PreparedStatement` [7] – the statement is cached and then the execution path is pre-determined on the database server allowing it to be executed multiple times in an efficient manner.
- `CallableStatement` [8] – used for executing stored procedures on the database.

Update statements such as INSERT, UPDATE and DELETE return an update count that indicates how many rows were affected in the database. These statements do not return any other information.

Query statements return a JDBC row result set. The row result set is used to walk over the result set. Individual columns in a row are retrieved either by name or by column number. There may be any number of rows in the result set. The row result set has metadata that describes the names of the columns and their types.

There is an extension to the basic JDBC API in the `javax.sql` [2].

JDBC connections are often managed via a connection pool rather than obtained directly from the driver. Examples of connection pools include BoneCP [9], C3P0 [10] and DBCP [11]

JDBC drivers

JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand.

Types

There are commercial and free drivers available for most relational database servers. These drivers fall into one of the following types:

- Type 1 that calls native code of the locally available ODBC driver.
- Type 2 that calls database vendor native library on a client side. This code then talks to database over network.
- Type 3, the pure-java driver that talks with the server-side middleware that then talks to database.
- Type 4, the pure-java driver that uses database native protocol.

There is also a type called internal JDBC driver, driver embedded with JRE in Java-enabled SQL databases. It's used for Java stored procedures. This does not belong to the above classification, although it would likely be either a type 2 or type 4 driver (depending on whether the database itself is implemented in Java or not). An example of this is the KPRB driver supplied with Oracle RDBMS. "jdbc:default:connection" is a relatively standard way of referring making such a connection (at least Oracle and Apache Derby support it). The distinction here is that the JDBC client is actually running as part of the database being accessed, so access can be made directly rather than through network protocols.

Sources

- SQLSummit.com publishes list of drivers, including JDBC drivers and vendors
- Oracle provides a list of some JDBC drivers and vendors [12]
- Simba Technologies ships an SDK for building custom JDBC Drivers for any custom/proprietary relational data source
- RSSBus Type 4 JDBC Drivers for applications, databases, and web services [13].
- DataDirect Technologies provides a comprehensive suite of fast Type 4 JDBC drivers for all major database they advertise as Type 5
- IDS Software provides a Type 3 JDBC driver for concurrent access to all major databases. Supported features include resultset caching, SSL encryption, custom data source, dbShield
- OpenLink Software ships JDBC Drivers for a variety of databases, including Bridges to other data access mechanisms (e.g., ODBC, JDBC) which can provide more functionality than the targeted mechanism
- JDBaccess is a Java persistence library for MySQL and Oracle which defines major database access operations in an easy usable API above JDBC
- JNetDirect provides a suite of fully Sun J2EE certified high performance JDBC drivers.
- HSQLDB is a RDBMS with a JDBC driver and is available under a BSD license.
- SchemaCrawler is an open source API that leverages JDBC, and makes database metadata available as plain old Java objects (POJOs)

References

[1] http://download.oracle.com/javase/7/docs/api/java/sql/package-summary.html
[2] http://download.oracle.com/javase/7/docs/api/javax/sql/package-summary.html
[3] JDBC API Specification Version: 4.0 (http://java.sun.com/products/jdbc/download.html#corespec40).
[4] JSR-000221 JDBC API Specification 4.1 (Maintenance Release) (http://jcp.org/aboutJava/communityprocess/mrel/jsr221/index.html)
[5] http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/jdbc_41.html
[6] http://download.oracle.com/javase/7/docs/api/java/sql/Statement.html
[7] http://download.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html
[8] http://download.oracle.com/javase/7/docs/api/java/sql/CallableStatement.html
[9] http://jolbox.com
[10] http://sourceforge.net/projects/c3p0
[11] http://commons.apache.org/dbcp
[12] http://devapp.sun.com/product/jdbc/drivers
[13] http://www.rssbus.com/jdbc/

External links

- Java SE 7 (http://download.oracle.com/javase/7/docs/) This documentation has examples where the JDBC resources are not closed appropriately (swallowing primary exceptions and being able to cause NullPointerExceptions) and has code prone to SQL injection[*citation needed*]
- `java.sql` (http://download.oracle.com/javase/7/docs/api/java/sql/package-summary.html) API Javadoc documentation
- `javax.sql` (http://download.oracle.com/javase/7/docs/api/javax/sql/package-summary.html) API Javadoc documentation
- O/R Broker (http://www.orbroker.org) Scala JDBC framework
- SqlTool (http://www.hsqldb.org/doc/2.0/util-guide/sqltool-chapt.html) Open source, command-line, generic JDBC client utility. Works with any JDBC-supporting database.
- JDBC URL Strings and related information of All Databases. (http://codeoftheday.blogspot.com/2012/12/java-database-connectivity-jdbc-url.html)

XQuery API for Java

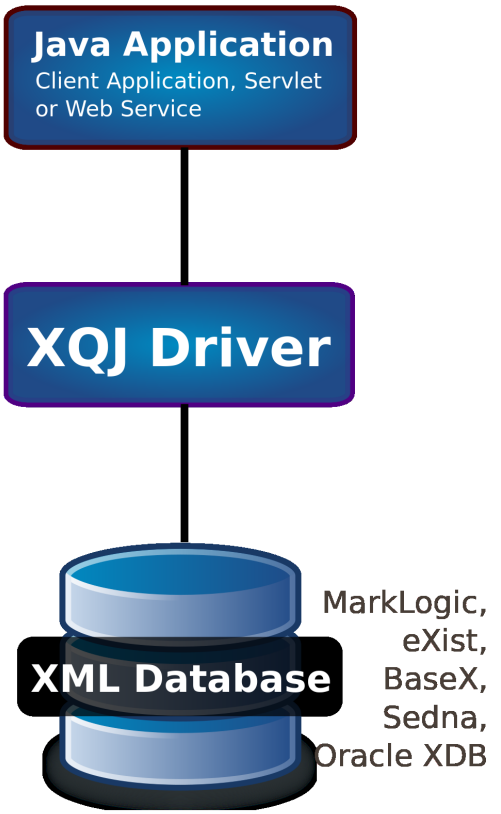
XQJ

Developer(s)	Java Community Process
Stable release	1.0 / June 24, 2009
Type	Data Access API
Website	JSR 225: XQuery API for Java [1]

XQuery API for Java (**XQJ**) refers to the common Java API for the W3C XQuery 1.0 specification.

The XQJ API enables Java programmers to execute XQuery against an XML data source (e.g. an XML database) while reducing or eliminating vendor lock in.

The XQJ API provides Java developers with an interface to the XQuery Data Model.[2] Its design is similar to the JDBC API which has a client/server feel and as such lends itself well to Server based XML Databases and less well to client-side XQuery processors, although the "connection" part is a very minor part of the entire API. Users of the XQJ API can bind Java values to XQuery expressions, preventing code injection attacks.[3] Also, multiple XQuery expressions can be executed as part of an atomic transaction.



General architecture of how an XQJ driver is used to communicate with an XML Database from Java Applications.

History and implementation

The XQuery API for Java was developed at the Java Community Process as JSR 225. It had some big technology backers such as Oracle,[4][5][6][7] IBM, BEA Systems,[8] Software AG,[9] Intel, Nokia and DataDirect.

Version 1.0 of the XQuery API for Java Specification was released on June 24, 2009,[10] along with JavaDocs, a reference implementation and a TCK (Technology Compatibility Kit) which implementing vendors must conform to.

The XQJ classes are contained in the Java package `javax.xml.xquery` [11]

Functionality

XQJ allows multiple implementations to exist and be used by the same application.

XQJ connections support creating and executing XQuery expressions. Expressions may be updating[12] and may include full text searches.[13] XQJ represents XQuery expressions using one of the following classes:

- XQExpression [14] – the expression is sent to the XQuery processor every time.
- XQPreparedExpression [15] – the expression is cached and the execution path is pre-determined allowing it to be executed multiple times in an efficient manner.

XQuery expressions return a result sequence of XDM items which in XQJ are represented through the XQResultSequence [16] interface. The programmer can use an XQResultSequence [16] to walk over individual XDM items in the result sequence. Each item in the sequence has XDM type information associated with it, such as its node type e.g. element(), document-node() or an XDM atomic type such as xs:string, xs:integer or xs:dateTime. XDM type information in XQJ can be retrieved via the XQItemType [17] interface.

Atomic XQuery items can be easily cast to Java primitives via XQItemAccessor [18] methods such as getByte() [19] and getFloat() [20]. Also XQuery items and sequences can be serialized to DOM Node [21], SAX ContentHandler [22], StAX XMLStreamReader [23] and the generic IO Reader [24] and InputStream [25] classes.

Examples

Basic Example

The following example illustrates creating a connection to an XML Database, submitting an XQuery expression, then processing the results in Java. Once all of the results have been processed, the connection is closed to free up all resources associated with it.

```
// Create a new connection to an XML database
XQConnection conn = vendorDataSource.getConnection("myUser", "myPassword");

XQExpression expr = conn.createExpression(); // Create a reusable XQuery Expression object

XQResultSequence result = expr.executeQuery(
  "for $n in fn:collection('catalog')//item " +
  "return fn:data($n/name)"); // execute an XQuery expression

// Process the result sequence iteratively
while (result.next()) {
  // Print the current item in the sequence
  System.out.println("Product name: " + result.getItemAsString(null));
}

// Free all resources created by the connection
conn.close();
```

Binding a value to an external variable

The following example illustrates how a Java value can be bound to an external variable in an XQuery expression. Assume that the connection `conn` already exists

```
XQExpression expr = conn.createExpression();

// The XQuery expression to be executed
String es = "declare variable $x as xs:integer external;" +
            " for $n in fn:collection('catalog')//item" +
            " where $n/price <= $x" +
            " return fn:data($n/name)";

// Bind a value (21) to an external variable with the QName x
expr.bindInt(new QName("x"), 21, null);

// Execute the XQuery expression
XQResultSequence result = expr.executeQuery(es);

// Process the result (sequence) iteratively
while (result.next()) {
  // Process the result ...
}
```

Default data type mapping

Mapping between Java and XQuery data types is largely flexible, however the XQJ 1.0 specification does have default mapping rules mapping data types when they are not specified by the user. These mapping rules bear great similarities to the mapping rules found in JAXB.

The following table illustrates the default mapping rules for when binding Java values to external variables in XQuery expressions.

Default conversion rules when mapping from Java data types to XQuery data types

Java Datatype	Default XQuery Data Type(s)
`boolean`	`xs:boolean`
`byte`	`xs:byte`
`byte[]`	`xs:hexBinary`
`double`	`xs:double`
`float`	`xs:float`
`int`	`xs:int`
`long`	`xs:long`
`short`	`xs:short`
`Boolean` [26]	`xs:boolean`
`Byte` [27]	`xs:byte`
`Float` [28]	`xs:float`

Double [29]	xs:double
Integer [30]	xs:int
Long [31]	xs:long
Short [32]	xs:short
String [33]	xs:string
BigDecimal [34]	xs:decimal
BigInteger [35]	xs:integer
Duration [36]	xs:dayTimeDuration if the Duration Object's state is xs:dayTimeDuration
	xs:yearMonthDuration if the Duration Object's state is xs:yearMonthDuration
	xs:duration if the Duration Object's state is xs:duration
XMLGregorianCalendar [37]	xs:date if the XMLGregorianCalendar Object's state is xs:date
	xs:dateTime if the XMLGregorianCalendar Object's state is xs:dateTime
	xs:gDay if the XMLGregorianCalendar Object's state is xs:gDay
	xs:gMonth if the XMLGregorianCalendar Object's state is xs:gMonth
	xs:gMonthDay if the XMLGregorianCalendar Object's state is xs:gMonthDay
	xs:gYear if the XMLGregorianCalendar Object's state is xs:gYear
	xs:gYearMonth if the XMLGregorianCalendar Object's state is xs:gYearMonth
	xs:time if the XMLGregorianCalendar Object's state is xs:time
QName [38]	xs:QName
Document [39]	document-node(element(*, xs:untyped))
DocumentFragment [40]	document-node(element(*, xs:untyped))
Element [41]	element(*, xs:untyped)
Attr [42]	attribute(*, xs:untypedAtomic)
Comment [43]	comment()
ProcessingInstruction [44]	processing-instruction()
Text [45]	text()

Known implementations

Native XML databases

The following is a list of Native XML Databases which are known to have XQuery API for Java implementations.

- MarkLogic[46]
- eXist[47]
- BaseX[48]
- Sedna[49]
- Oracle XDB [50][51]
- Tamino[52]
- TigerLogic

Relational databases

DataDirect provide XQJ adapters for relational databases, by translating XQuery code into SQL on the fly, then converting SQL result sets into a format suitable for XQJ to process further. The following is a couple of known implementations.

- Oracle DB (Not XDB)
- IBM DB2
- Microsoft SQL Server
- Sybase ASE
- Informix
- MySQL
- PostgreSQL

Client-side implementations

The following is a list of client-side XQuery processors which provide an XQuery API for Java interface.

- Saxon XSLT and XQuery processor
- Zorba[53]
- MXQuery
- Oracle XQuery Processor [54]

References

[1] http://jcp.org/en/jsr/detail?id=225
[2] XQuery 1.0 and XPath 2.0 Data Model (XDM) (http://www.w3.org/TR/xpath-datamodel/)
[3] Binding Java Variables (http://www.cfoster.net/articles/xqj-tutorial/binding-java-variables.xml)
[4] Querying XML: XQuery, XPath, and SQL/XML in context - Jim Melton and Stephen Buxton. ISBN 978-1558607118
[5] XQJ - XQuery Java API is Completed, Marc Van Cappellen, Zhen Hua Liu, Jim Melton and Maxim Orgiyan (http://www.sigmod.org/publications/sigmod-record/0912/p07.article.cappellen.pdf)
[6] IBM and Oracle Submit XQuery API for Java (XQJ) Java Specification Request. (http://xml.coverpages.org/ni2003-06-12-b.html)
[7] An Early Look at XQuery API for Java (XQJ) - Andrew Eisenberg, IBM and Jim Melton, Oracle (http://www.sigmod.org/publications/sigmod-record/0406/JimAndrew.pdf)
[8] The BEA Streaming XQuery Processor (http://www.cfoster.net/pdf/reference/10.1.1.92.2337.pdf#page=17)
[9] XQJ Interface for Tamino Native XML Database (http://documentation.softwareag.com/webmethods/wmsuites/wmsuite8-2_ga/CentraSite/8-2-SP1_CentraSite/dg-xqj/overview.htm)
[10] JSR-000225 XQuery API for Java (Final Release) (http://jcp.org/aboutJava/communityprocess/final/jsr225/index.html)
[11] http://xqj.net/javadoc/
[12] XQuery Update Facility
[13] XQuery Full Text (http://www.w3.org/TR/xpath-full-text-10/)
[14] http://xqj.net/javadoc/javax/xml/xquery/XQExpression.html

[15] http://xqj.net/javadoc/javax/xml/xquery/XQPreparedExpression.html
[16] http://xqj.net/javadoc/javax/xml/xquery/XQResultSequence.html
[17] http://xqj.net/javadoc/javax/xml/xquery/XQItemType.html
[18] http://xqj.net/javadoc/javax/xml/xquery/XQItemAccessor.html
[19] http://xqj.net/javadoc/javax/xml/xquery/XQItemAccessor.html#getByte()
[20] http://xqj.net/javadoc/javax/xml/xquery/XQItemAccessor.html#getFloat()
[21] http://download.oracle.com/javase/7/docs/api/org/w3c/dom/Node.html
[22] http://download.oracle.com/javase/7/docs/api/org/xml/sax/ContentHandler.html
[23] http://download.oracle.com/javase/7/docs/api/javax/xml/stream/XMLStreamReader.html
[24] http://download.oracle.com/javase/7/docs/api/java/io/Reader.html
[25] http://download.oracle.com/javase/7/docs/api/java/io/InputStream.html
[26] http://download.oracle.com/javase/7/docs/api/java/lang/Boolean.html
[27] http://download.oracle.com/javase/7/docs/api/java/lang/Byte.html
[28] http://download.oracle.com/javase/7/docs/api/java/lang/Float.html
[29] http://download.oracle.com/javase/7/docs/api/java/lang/Double.html
[30] http://download.oracle.com/javase/7/docs/api/java/lang/Integer.html
[31] http://download.oracle.com/javase/7/docs/api/java/lang/Long.html
[32] http://download.oracle.com/javase/7/docs/api/java/lang/Short.html
[33] http://download.oracle.com/javase/7/docs/api/java/lang/String.html
[34] http://download.oracle.com/javase/7/docs/api/java/math/BigDecimal.html
[35] http://download.oracle.com/javase/7/docs/api/java/math/BigInteger.html
[36] http://download.oracle.com/javase/7/docs/api/javax/xml/datatype/Duration.html
[37] http://download.oracle.com/javase/7/docs/api/javax/xml/datatype/XMLGregorianCalendar.html
[38] http://download.oracle.com/javase/7/docs/api/javax/xml/namespace/QName.html
[39] http://download.oracle.com/javase/7/docs/api/org/w3c/dom/Document.html
[40] http://download.oracle.com/javase/7/docs/api/org/w3c/dom/DocumentFragment.html
[41] http://download.oracle.com/javase/7/docs/api/org/w3c/dom/Element.html
[42] http://download.oracle.com/javase/7/docs/api/org/w3c/dom/Attr.html
[43] http://download.oracle.com/javase/7/docs/api/org/w3c/dom/Comment.html
[44] http://download.oracle.com/javase/7/docs/api/org/w3c/dom/ProcessingInstruction.html
[45] http://download.oracle.com/javase/7/docs/api/org/w3c/dom/Text.html
[46] MarkLogic XQJ API (http://xqj.net/marklogic)
[47] eXist XQJ API (http://xqj.net/exist)
[48] BaseX XQJ API (http://xqj.net/basex)
[49] Sedna XQJ API (http://xqj.net/sedna)
[50] http://www.oracle.com/technetwork/database-features/xmldb/overview/index.html
[51] Oracle XML DB Support for XQJ (http://docs.oracle.com/cd/E16655_01/appdev.121/e17604/adx_j_xqjxdb.htm#ADXDK136)
[52] Software AG - Working with the CentraSite XQJ Interface (http://documentation.softwareag.com/webmethods/wmsuites/wmsuite8-2_ga/CentraSite/8-2-SP1_CentraSite/dg-xqj/working_xqjdriver.htm)
[53] Zorba 2.5 ships with a long awaited XQJ binding, 14th June 2012 (http://www.zorba-xquery.com/html/entry/2012/06/14/Zorba_25)
[54] Oracle XML Developer's Kit (XDK) provides a standalone XQuery 1.0 processor for use by Java applications. (http://docs.oracle.com/cd/E16655_01/appdev.121/e17604/adx_j_xqj.htm#ADXDK99930)

External links

- Javadoc for XQJ (http://xqj.net/javadoc/)
- XQJ Tutorial (http://www.cfoster.net/articles/xqj-tutorial/)
- Building Bridges from Java to XQuery, Charles Foster. XML Prague 2012 (http://archive.xmlprague.cz/2012/files/xmlprague-2012-proceedings.pdf#page=197) (Prezi Presentation (http://prezi.com/lviyahwtaxge/building-bridges-from-java-to-xquery/))
- Java Integration of XQuery, Hans-J‘rgen Rennau. Balisage 2010 (http://www.balisage.net/Proceedings/vol5/html/Rennau01/BalisageVol5-Rennau01.html)
- Orbeon Forms using XQJ (http://wiki.orbeon.com/forms/doc/developer-guide/processors-xquery-generator#TOC-XQuery-processor-implementations)
- Spring Integration XQuery Support (https://github.com/SpringSource/spring-integration-extensions/tree/master/spring-integration-xquery)

€ XQS: XQuery for Scala (Sits on top of XQJ) (https://github.com/fancellu/xqs)

ODBC

In computing, **ODBC** (**Open Database Connectivity**) is a standard programming language middleware API for accessing database management systems (DBMS). The designers of ODBC aimed to make it independent of database systems and operating systems; an application written using ODBC can be ported to other platforms, both on the client and server side, with few changes to the data access code.

ODBC accomplishes DBMS independence by using an **ODBC driver** as a translation layer between the application and the DBMS. The application uses ODBC functions through an **ODBC driver manager** with which it is linked, and the driver passes the query to the DBMS. An ODBC driver can be thought of as analogous to a printer or other driver, providing a standard set of functions for the application to use, and implementing DBMS-specific functionality. An application that can use ODBC is referred to as "ODBC-compliant". Any ODBC-compliant application can access any DBMS for which a driver is installed. Drivers exist for all major DBMSs, many other data sources like address book systems and Microsoft Excel, and even for text or CSV files.

ODBC was originally developed by Microsoft during the early 1990s, and became the basis for the Call Level Interface (CLI) standardized by SQL Access Group in the Unix and mainframe world. ODBC retained a number of features that were removed as part of the CLI effort. Full ODBC was later ported back to those platforms, and became a de facto standard considerably better known than CLI. The CLI remains similar to ODBC, and applications can be ported from one platform to the other with few changes.

History

Prior to ODBC

The introduction of the mainframe-based relational database during the 1970s led to a proliferation of data access methods. Generally these systems operated hand-in-hand with a simple command processor that allowed the user to type in English-like commands, and receive output. The best-known examples are SEQUEL from IBM and QUEL from the Ingres project. These systems may or may not allow other applications to access the data directly, and those that did used a wide variety of methodologies. The introduction of SQL aimed to solve the problem of *language* standardization, although substantial differences in implementation remained.

Additionally, since the SQL language had only rudimentary programming features, it was often desired to use SQL within a program written in another language, say Fortran or C. This led to the concept of Embedded SQL, which allowed SQL code to be "embedded" within another language. For instance, a SQL statement like `SELECT * FROM city` could be inserted as text within C source code, and during compilation it would be converted into a custom format that directly called a function within a library that would pass the statement into the SQL system. Results returned from the statements would be interpreted back into C data formats like `char *` using similar library code.

There were a number of problems with the Embedded SQL approach. Like the different varieties of SQL, the Embedded SQL's that used them varied widely, not only from platform to platform, but even across languages on a single platform - a system that allowed calls into IBM's DB2 would look entirely different than one that called into their own SQL/DSWikipedia:Disputed statement. Another key problem to the Embedded SQL concept was that the SQL code could only be changed in the program's source code, so that even small changes to the query required considerable programmer effort to modify. The SQL market referred to this as "static SQL", as opposed to "dynamic SQL" which could be changed at any time - like the command-line interfaces that shipped with almost all SQL systems, or a programming interface that left the SQL as plain text until it was called. Dynamic SQL systems

became a major focus for SQL vendors during the 1980s.

Older mainframe databases, and the newer microcomputer based systems that were based on them, generally did not have a SQL-like command processor between the user and the database engine. Instead, the data was accessed directly by the program - a programming library in the case of large mainframe systems, or a command line interface or interactive forms system in the case of dBASE and similar applications. Data from dBASE could not generally be accessed directly by other programs running on the machine. Those programs may be given a way to access this data, often through libraries, but it would not work with any other database engine, or even different databases in the same engine. In effect, all such systems were static, which presented considerable problems.

Early efforts

By the mid-1980s the rapid improvement in microcomputers, and especially the introduction of the graphical user interface and data-rich application programs like Lotus 1-2-3 led to an increasing interest in using personal computers as the client-side platform of choice in client-server computing. Under this model, large mainframes and minicomputers would be used primarily to serve up data over local area networks to microcomputers that would interpret, display and manipulate that data. For this model to work, a data access standard was a requirement - in the mainframe world it was highly likely that all of the computers in a shop were from a single vendor and clients were computer terminals talking directly to them, but in the micro world there was no such standardization and any client might access any server using any networking system.

By the late 1980s there were a number of efforts underway to provide an abstraction layer for this purpose. Some of these were mainframe related, designed to allow programs running on those machines to translate between the variety of SQL's and provide a single common interface which could then be called by other mainframe or microcomputer programs. These solutions included IBM's Distributed Relational Database Architecture (DRDA) and Apple Computer's Data Access Language. Much more common, however, were systems that ran entirely on microcomputers, including a complete protocol stack that included any required networking or file translation support.

One of the early examples of such a system was Lotus Development's DataLens, initially known as Blueprint. Blueprint, developed for 1-2-3, supported a variety of data sources, including SQL/DS, DB2, FOCUS and a variety of similar mainframe systems, as well as microcomputer systems like dBase and the early Microsoft/Ashton-Tate efforts that would eventually develop into Microsoft SQL Server.[1] Unlike the later ODBC, Blueprint was a purely code-based system, lacking anything approximating a command language like SQL. Instead, programmers used data structures to store the query information, constructing a query by linking many of these structures together. Lotus referred to these compound structures as "query trees".[2]

Around the same time, an industry team including members from Sybase, Tandem Computers and Microsoft were working on a standardized dynamic SQL concept. Much of the system was based on Sybase's DB-Library system, with the Sybase-specific sections removed and several additions to support other platforms.[3] DB-Library was aided by an industry-wide move from library systems that were tightly linked to a particular language, to library systems that were provided by the operating system and required the languages on that platform to conform to its standards. This meant that a single library could be used with (potentially) any programming language on a given platform.

The first draft of the **Microsoft Data Access API** was published in April 1989, about the same time as Lotus' announcement of Blueprint.[4] In spite of Blueprint's great lead - it was running when MSDA was still a paper project - Lotus eventually joined the MSDA efforts as it became clear that SQL would become the de facto database standard.[2] After considerable industry input, in the summer of 1989 the standard became **SQL Connectivity**, or **SQLC** for short,.[5]

SAG and CLI

In 1988 a number of vendors, mostly from the Unix and database communities, formed the SQL Access Group (SAG) in an effort to produce a single basic standard for the SQL language. At the first meeting there was considerable debate over whether or not the effort should work solely on the SQL language itself, or attempt a wider standardization which included a dynamic SQL language-embedding system as well, what they called a Call Level Interface (CLI).[6] While attending the meeting with an early draft of what was then still known as MS Data Access, Kyle Geiger of Microsoft invited Jeff Balboni and Larry Barnes of Digital Equipment Corporation (DEC) to join the SQLC meetings as well. SQLC was a potential solution to the call for the CLI, which was being led by DEC.

The new SQLC "gang of four", MS, Lotus, DEC and Sybase, brought an updated version of SQLC to the next SAG meeting in June 1990.[7] The SAG responded by opening the standard effort to any competing design, but of the many proposals, only Oracle Corp had a system that presented serious competition. In the end, SQLC won the votes and became the draft standard, but only after large portions of the API were removed - the standards document was trimmed from 120 pages to 50 during this time. It was also during this period that the name Call Level Interface was formally adopted.[7] In 1995 SQL/CLI became part of the international SQL standard, ISO/IEC 9075-3.[8] The SAG itself was taken over by the X/Open group in 1996, and, over time, became part of The Open Group's Common Application Environment.

MS continued working with the original SQLC standard, retaining many of the advanced features that were removed from the CLI version. These included features like scrollable cursors, and metadata information queries. The commands in the API were split into groups; the Core group was identical to the CLI, the Level 1 extensions were commands that would be easy to implement in drivers, while Level 2 commands contained the more advanced features like cursors. A proposed standard was released in December 1991, and industry input was gathered and worked into the system through 1992, resulting in yet another name change to **ODBC**.[9]

JET and ODBC

During this time, Microsoft was in the midst of developing their Jet database system. Jet combined three primary subsystems; an ISAM-based database engine (also known as "Jet", confusingly), a C-based interface allowing applications to access that data, and a selection of driver DLLs that allowed the same C interface to redirect input and output to other ISAM-based databases, like Paradox and xBase. Jet allowed programmers to use a single set of calls to access common microcomputer databases in a fashion similar to Blueprint (by this point known as DataLens). However, Jet did not use SQL; like DataLens, the interface was in C and consisted of data structures and function calls.

The SAG standardization efforts presented an opportunity for Microsoft to adapt their Jet system to the new CLI standard. This would not only make Windows a premier platform for CLI development, but also allow users to use SQL to access both Jet and other databases as well. What was missing was the SQL parser that could convert those calls from their text form into the C-interface used in Jet. To solve this, MS partnered with PageAhead Software to use their existing query processor, "SIMBA". SIMBA was used as a parser above Jet's C library, turning Jet into an SQL database. And because Jet could forward those C-based calls to other databases, this also allowed SIMBA to query other systems. Microsoft included drivers for Excel to turn its spreadsheet documents into SQL-accessible database tables.

Release and continued development

ODBC 1.0 was released in September 1992. At the time, there was little direct support for SQL databases (as opposed to ISAM), and early drivers were noted for poor performance. Some of this was unavoidable due to the path that the calls took through the Jet-based stack; ODBC calls to SQL databases were first converted from SIMBA's SQL dialect to Jet's internal C-based format, then passed to a driver for conversion back into SQL calls for the database. Digital Equipment and Oracle both contracted Simba to develop drivers for their databases as well.[10]

Meanwhile the CLI standard effort dragged on, and it was not until March 1995 that the definitive version was finalized. By this time Microsoft had already granted Visigenic Software a source code license to develop ODBC on non-Windows platforms. Visigenic ported ODBC to a wide variety of Unix platforms, where ODBC quickly became the de facto standard.[11] "Real" CLI is rare today. The two systems remain similar, and many applications can be ported from ODBC to CLI with few or no changes.[12]

Over time, database vendors took over the driver interfaces and provided direct links to their products. Skipping the intermediate conversions to and from Jet or similar wrappers often resulted in higher performance. However, by this time Microsoft had changed focus to their OLE DB concept, which provided direct access to a wider variety of data sources from address books to text files. Several new systems followed which further turned their attention from ODBC, including DAO, ADO and ADO.net, which interacted more or less with ODBC over their lifetimes.

As Microsoft turned its attention away from working directly on ODBC, the Unix world was increasingly embracing it. This was propelled by two changes within the market, the introduction of GUIs like GNOME that provided the need for access to these sources in non-text form, and the emergence of open software database systems like PostgreSQL and MySQL, initially under Unix. The later adoption of ODBC by Apple for Mac OS X 10.4 using the standard Unix-side iODBC package further cemented ODBC as the standard for cross-platform data access.

Sun Microsystems used the ODBC system as the basis for their own open standard, JDBC. In most ways, JDBC can be considered a version of ODBC for the Java programming language as opposed to C. JDBC-to-ODBC "bridges" allow JDBC programs to access data sources through ODBC drivers on platforms lacking a native JDBC driver, although these are now relatively rare.

ODBC today

ODBC remains largely universal today, with drivers available for most platforms and most databases. It is not uncommon to find ODBC drivers for database engines that are meant to be embedded, like SQLite, as a way to allow existing tools to act as front-ends to these engines for testing and debugging.[13]

However, the rise of thin client computing using HTML as an intermediate format has reduced the need for ODBC. Many web development platforms contain direct links to target databases - MySQL being particularly common. In these scenarios, there is no direct client-side access nor multiple client software systems to support, everything goes through the programmer-supplied HTML application. The virtualization that ODBC offers is no longer a strong requirement, and development of ODBC is no longer as active as it once was.

Version history

Version history:

- 1.0: released in September 1992
- 2.0: ca 1994
- 2.5
- 3.0: ca 1995, John Goodson of Intersolv and Frank Pellow and Paul Cotton of IBM provided significant input to OBDC 3.0[14]
- 3.5: ca 1997
- 3.8: ca 2009, with Windows 7

Drivers and Managers

Drivers

ODBC is based on the device driver model, where the driver encapsulates the logic needed to convert a standard set of commands and functions into the specific calls required by the underlying system. For instance, a printer driver presents a standard set of printing commands, the API, to applications using the printing system. Calls made to those APIs are converted by the driver into the format used by the actual hardware, say PostScript or PCL.

In the case of ODBC, the drivers encapsulate a number of functions that can be broken down into several broad categories. One set of functions is primarily concerned with finding, connecting to and disconnecting from the DBMS that driver talks to. A second set is used to send SQL commands from the ODBC system to the DBMS, converting or interpreting any commands that are not supported internally. For instance, a DBMS that does not support cursors can emulate this functionality in the driver. Finally, another set of commands, mostly used internally, is used to convert data from the DBMS's internal formats to a set of standardized ODBC formats, which are based on the C language formats.

An ODBC driver enables an ODBC-compliant application to use a *data source*, normally a DBMS. Some non-DBMS drivers exist, for such data sources as CSV files, by implementing a small DBMS inside the driver itself. ODBC drivers exist for most DBMSs, including Oracle, PostgreSQL, MySQL, Microsoft SQL Server (but not for the Compact aka CE edition), Sybase ASE, and DB2. Because different technologies have different capabilities, most ODBC drivers do not implement all functionality defined in the ODBC standard. Some drivers offer extra functionality not defined by the standard.

Driver Manager

Device drivers are normally enumerated, set up and managed by a separate Manager layer, which may provide additional functionality. For instance, printing systems often include functionality to provide spooling functionality on top of the drivers, providing print spooling for any supported printer.

In ODBC the Driver Manager (DM) provides these features. The DM can enumerate the installed drivers and present this as a list, often in a GUI-based form.

But more important to the operation of the ODBC system is the DM's concept of **Data Source Name**s, or **DSN**. DSNs collect additional information needed to connect to a *particular* data source, as opposed to the DBMS itself. For instance, the same MySQL driver can be used to connect to any MySQL server, but the connection information to connect to a local private server is different than the information needed to connect to an internet-hosted public server. The DSN stores this information in a standardized format, and the DM provides this to the driver during connection requests. The DM also includes functionality to present a list of DSNs using human readable names, and to select them at run-time to connect to different resources.

The DM also includes the ability to save partially complete DSN's, with code and logic to ask the user for any missing information at runtime. For instance, a DSN can be created without a required password. When an ODBC application attempts to connect to the DBMS using this DSN, the system will pause and ask the user to provide the password before continuing. This frees the application developer from having to create this sort of code, as well as having to know which questions to ask. All of this is included in the driver and the DSNs.

Bridging configurations

A *bridge* is a special kind of driver: a driver that uses another driver-based technology.

JDBC-ODBC bridges

A JDBC-ODBC bridge consists of a JDBC driver which employs an ODBC driver to connect to a target database. This driver translates JDBC method calls into ODBC function calls. Programmers usually use such a bridge when a particular database lacks a JDBC driver. Sun Microsystems included one such bridge in the JVM, but viewed it as a stop-gap measure while few JDBC drivers existed. Sun never intended its bridge for production environments, and generally recommends against its use. As of 2008[15] independent data-access vendors deliver JDBC-ODBC bridges which support current standards for both mechanisms, and which far outperform the JVM built-in.[*citation needed*]

ODBC-JDBC bridges

An ODBC-JDBC bridge consists of an **ODBC** driver which uses the services of a JDBC driver to connect to a database. This driver translates ODBC function-calls into JDBC method-calls. Programmers usually use such a bridge when they lack an ODBC driver for a particular database but have access to a JDBC driver.

OLE DB

Microsoft provides an OLE DB-ODBC bridge for simplifying development in COM aware languages (e.g. Visual Basic). This bridge forms part of the MDAC system component bundle, together with other database drivers.

References

Citations

[1] Evan McGlinn, Blueprint Lets 1-2-3 Access Outside Data" (http://books.google.ca/books?id=6D4EAAAAMBAJ), *InfoWorld*, 4 April 1988, p. 1, 69
[2] Geiger 1995, p.ƒ65.
[3] Geiger 1995, p.ƒ86-87.
[4] Geiger 1995, p.ƒ56.
[5] Geiger 1995, p.ƒ106.
[6] Geiger 1995, p.ƒ165.
[7] Geiger 1995, p.ƒ186-187.
[8] ISO/IEC 9075-3 -- Information technology -- Database languages -- SQL -- Part 3: Call-Level Interface (SQL/CLI)
[9] Geiger 1995, p.ƒ203.
[10] "Our History" (http://www.simba.com/simba-history.htm), Simba Technologies
[11] Roger Sippl, "SQL Access Group's Call-Level Interface" (http://www.drdobbs.com/sql-access-groups-call-level-interface/184410032), Dr. Dobbs, 1 February 1996
[12] "Similarities and differences between ODBC and CLI" (http://publib.boulder.ibm.com/infocenter/iisclzos/v9r5/index.jsp?topic=/com.ibm.swg.im.iis.fed.classic.clientsref.doc/topics/iiyfcodbcclisimdiff.html), InfoSphere Classic documentation, IBM, 26 September 2008
[13] Christian Werner, "SQLite ODBC Driver" (http://www.ch-werner.de/sqliteodbc/)
[14] Microsoft Corporation. Microsoft ODBC 3.0 Programmer's Reference and SDK Guide, Volume 1. Microsoft Press. February 1997. (ISBN13: 9781572315167)
[15] http://en.wikipedia.org/w/index.php?title=ODBC&action=edit

Bibliography

€ Kyle Geiger, "Inside ODBC" (http://books.google.ca/books?id=G-ZQAAAAMAAJ&), Microsoft Press, 1995

External links

- Microsoft ODBC Overview (http://support.microsoft.com/kb/110093)
- List of ODBC Drivers at databasedrivers.com (http://www.databasedrivers.com/odbc/)
- List of ODBC Drivers at SQLSummit.com (http://www.SQLSummit.com/ODBCVend.htm)
- OS400 and i5OS ODBC Administration (http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/rzaii/rzaiiodbcadm.htm)
- Presentation slides from www.roth.net (http://www.roth.net/perl/odbc/conf/sld002.htm)
- Early ODBC White Paper (http://www.openlinksw.com/info/docs/odbcwhp/tableof.htm)
- Microsoft ODBC & Data Access APIs History Article (http://blogs.msdn.com/data/archive/2006/12/05/data-access-api-of-the-day-part-i.aspx)

Query language

Query languages are computer languages used to make queries into databases and information systems.

Broadly, query languages can be classified according to whether they are database query languages or information retrieval query languages. The difference is that a database query language attempts to give factual answers to factual questions, while an information retrieval query language attempts to find documents containing information that is relevant to an area of inquiry.

Examples include:

- .QL is a proprietary object-oriented query language for querying relational databases; successor of Datalog;
- PL/SQL is Oracle Corporation's procedural extension language for SQL and the Oracle relational database.
- Contextual Query Language (CQL) a formal language for representing queries to information retrieval systems such as web indexes or bibliographic catalogues.
- CQLF (CODASYL Query Language, Flat) is a query language for CODASYL-type databases;
- Concept-Oriented Query Language (COQL) is used in the concept-oriented model (COM). It is based on a novel data modeling construct, concept, and uses such operations as projection and de-projection for multi-dimensional analysis, analytical operations and inference;
- DMX is a query language for Data Mining models;
- Datalog is a query language for deductive databases;
- F-logic is a declarative object-oriented language for deductive databases and knowledge representation.
- Gellish English is a language that can be used for queries in Gellish English Databases, for dialogues (requests and responses) as well as for information modeling and knowledge modeling;[1]
- HTSQL is a query language that translates HTTP queries to SQL;
- ISBL is a query language for PRTV, one of the earliest relational database management systems;
- LINQ query-expressions is a way to query various data sources from .NET languages
- LDAP is an application protocol for querying and modifying directory services running over TCP/IP;
- MQL is a cheminformatics query language for a substructure search allowing beside nominal properties also numerical properties;
- MDX is a query language for OLAP databases;
- OQL is Object Query Language;
- OCL (Object Constraint Language). Despite its name, OCL is also an object query language and an OMG standard;
- OPath, intended for use in querying WinFS *Stores*;
- OttoQL, intended for querying tables, XML, and databases;

- Poliqarp Query Language is a special query language designed to analyze annotated text. Used in the Poliqarp search engine;
- QUEL is a relational database access language, similar in most ways to SQL;
- RDQL is a RDF query language;
- SMARTS is the cheminformatics standard for a substructure search;
- SPARQL is a query language for RDF graphs;
- SPL is a search language for machine-generated big data, based upon Unix Piping and SQL.
- SQL is a well known query language and Data Manipulation Language for relational databases;
- SuprTool is a proprietary query language for SuprTool, a database access program used for accessing data in *Image/SQL* (formerly TurboIMAGE) and Oracle databases;
- TMQL Topic Map Query Language is a query language for Topic Maps;
- **Tutorial D** is a query language for truly relational database management systems (TRDBMS);
- XQuery is a query language for XML data sources;
- XPath is a declarative language for navigating XML documents;
- XSPARQL is an integrated query language combining XQuery with SPARQL to query both XML and RDF data sources at once;
- YQL is an SQL-like query language created by Yahoo!

References

[1] http://gellish.wiki.sourceforge.net/Querying+a+Gellish+English+database

Query optimization

Query optimization is a function of many relational database management systems. The **query optimizer** attempts to determine the most efficient way to execute a given query by considering the possible query plans.

Generally, the query optimizer cannot be accessed directly by users: once queries are submitted to database server, and parsed by the parser, they are then passed to the query optimizer where optimization occurs. However, some database engines allow guiding the query optimizer with hints.

A query is a request for information from a database. It can be as simple as "finding the address of a person with SS# 123-45-6789," or more complex like "finding the average salary of all the employed married men in California between the ages 30 to 39, that earn less than their wives." Queries results are generated by accessing relevant database data and manipulating it in a way that yields the requested information. Since database structures are complex, in most cases, and especially for not-very-simple queries, the needed data for a query can be collected from a database by accessing it in different ways, through different data-structures, and in different orders. Each different way typically requires different processing time. Processing times of a same query may have large variance, from a fraction of a second to hours, depending on the way selected. The purpose of query optimization, which is an automated process, is to find the way to process a given query in minimum time. The large possible variance in time justifies performing query optimization, though finding the exact optimal way to execute a query, among all possibilities, is typically very complex, time consuming by itself, may be too costly, and often practically impossible. Thus query optimization typically tries to approximate the optimum by comparing several common-sense alternatives to provide in a reasonable time a "good enough" plan which typically does not deviate much from the best possible result.

General considerations

There is a trade-off between the amount of time spent figuring out the best query plan and the quality of the choice; the optimizer may not choose the best answer on its own. Different qualities of database management systems have different ways of balancing these two. Cost-based query optimizers evaluate the resource footprint of various query plans and use this as the basis for plan selection. These assign an estimated "cost" to each possible query plan, and choose the plan with the smallest cost. Costs are used to estimate the runtime cost of evaluating the query, in terms of the number of I/O operations required, CPU path length, amount of disk buffer space, disk storage service time, and interconnect usage between units of parallelism, and other factors determined from the data dictionary. The set of query plans examined is formed by examining the possible access paths (e.g., primary index access, secondary index access, full file scan) and various relational table join techniques (e.g., merge join, hash join, product join). The search space can become quite large depending on the complexity of the SQL query. There are two types of optimization. These consist of logical optimization which generates a sequence of relational algebra to solve the query. In addition there is physical optimization which is used to determine the means of carrying out each operation.

Implementation

Most query optimizers represent query plans as a tree of "plan nodes". A plan node encapsulates a single operation that is required to execute the query. The nodes are arranged as a tree, in which intermediate results flow from the bottom of the tree to the top. Each node has zero or more child nodes *ƒ* those are nodes whose output is fed as input to the parent node. For example, a join node will have two child nodes, which represent the two join operands, whereas a sort node would have a single child node (the input to be sorted). The leaves of the tree are nodes which produce results by scanning the disk, for example by performing an index scan or a sequential scan.

Join ordering

The performance of a query plan is determined largely by the order in which the tables are joined. For example, when joining 3 tables A, B, C of size 10 rows, 10,000 rows, and 1,000,000 rows, respectively, a query plan that joins B and C first can take several orders-of-magnitude more time to execute than one that joins A and C first. Most query optimizers determine join order via a dynamic programming algorithm pioneered by IBM's System R database project [*citation needed*]. This algorithm works in two stages:

1. First, all ways to access each relation in the query are computed. Every relation in the query can be accessed via a sequential scan. If there is an index on a relation that can be used to answer a predicate in the query, an index scan can also be used. For each relation, the optimizer records the cheapest way to scan the relation, as well as the cheapest way to scan the relation that produces records in a particular sorted order.
2. The optimizer then considers combining each pair of relations for which a join condition exists. For each pair, the optimizer will consider the available join algorithms implemented by the DBMS. It will preserve the cheapest way to join each pair of relations, in addition to the cheapest way to join each pair of relations that produces its output according to a particular sort order.
3. Then all three-relation query plans are computed, by joining each two-relation plan produced by the previous phase with the remaining relations in the query.

In this manner, a query plan is eventually produced that joins all the queries in the relation. Note that the algorithm keeps track of the sort order of the result set produced by a query plan, also called an *interesting order*. During dynamic programming, one query plan is considered to beat another query plan that produces the same result, only if they produce the same sort order. This is done for two reasons. First, a particular sort order can avoid a redundant sort operation later on in processing the query. Second, a particular sort order can speed up a subsequent join because it clusters the data in a particular way.

Historically, System-R derived query optimizers would often only consider *left-deep* query plans, which first join two base tables together, then join the intermediate result with another base table, and so on. This heuristic reduces the number of plans that need to be considered (n! instead of 4^n), but may result in not considering the optimal query plan. This heuristic is drawn from the observation that join algorithms such as nested loops only require a single tuple (aka row) of the outer relation at a time. Therefore, a left-deep query plan means that fewer tuples need to be held in memory at any time: the outer relation's join plan need only be executed until a single tuple is produced, and then the inner base relation can be scanned (this technique is called "pipelining").

Subsequent query optimizers have expanded this plan space to consider "bushy" query plans, where both operands to a join operator could be intermediate results from other joins. Such bushy plans are especially important in parallel computers because they allow different portions of the plan to be evaluated independently.

Query planning for nested SQL queries

A SQL query to a modern relational DBMS does more than just selections and joins. In particular, SQL queries often nest several layers of SPJ blocks (Select-Project-Join), by means of group by, exists, and not exists operators. In some cases such nested SQL queries can be flattened into a select-project-join query, but not always. Query plans for nested SQL queries can also be chosen using the same dynamic programming algorithm as used for join ordering, but this can lead to an enormous escalation in query optimization time. So some database management systems use an alternative rule-based approach that uses a query graph model.

Cost estimation

One of the hardest problems in query optimization is to accurately estimate the costs of alternative query plans. Optimizers cost query plans using a mathematical model of query execution costs that relies heavily on estimates of the cardinality, or number of tuples, flowing through each edge in a query plan. Cardinality estimation in turn depends on estimates of the selection factor[1] of predicates in the query. Traditionally, database systems estimate selectivities through fairly detailed statistics on the distribution of values in each column, such as histograms. This technique works well for estimation of selectivities of individual predicates. However many queries have conjunctions of predicates such as `select count(*) from R where R.make='Honda' and R.model='Accord'`. Query predicates are often highly correlated (for example, `model='Accord'` implies `make='Honda'`), and it is very hard to estimate the selectivity of the conjunct in general. Poor cardinality estimates and uncaught correlation are one of the main reasons why query optimizers pick poor query plans. This is one reason why a database administrator should regularly update the database statistics, especially after major data loads/unloads.

References

- Chaudhuri, Surajit (1998). "An Overview of Query Optimization in Relational Systems" [2]. *Proceedings of the ACM Symposium on Principles of Database Systems*. pp. pages 34–43. doi:10.1145/275487.275492 [3].
- Ioannidis, Yannis (March 1996). "Query optimization" [4]. *ACM Computing Surveys* **28** (1): 121–123. doi:10.1145/234313.234367 [5].
- Selinger, P. G.; Astrahan, M. M.; Chamberlin, D. D.; Lorie, R. A.; Price, T. G. (1979). "Access Path Selection in a Relational Database Management System". *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. pp. 23–34. doi:10.1145/582095.582099 [6]. ISBN 089791001X

References

[1] http://toolserver.org/%7Edispenser/cgi-bin/dab_solver.py?page=Query_optimization&editintro=Template:Disambiguation_needed/editintro&client=Template:Dn
[2] http://citeseer.ist.psu.edu/chaudhuri98overview.html
[3] http://dx.doi.org/10.1145%2F275487.275492
[4] http://citeseer.ist.psu.edu/487912.html
[5] http://dx.doi.org/10.1145%2F234313.234367
[6] http://dx.doi.org/10.1145%2F582095.582099

Query plan

A **query plan** (or **query execution plan**) is an ordered set of steps used to access data in a SQL relational database management system. This is a specific case of the relational model concept of access plans.

Since SQL is declarative, there are typically a large number of alternative ways to execute a given query, with widely varying performance. When a query is submitted to the database, the query optimizer evaluates some of the different, correct possible plans for executing the query and returns what it considers the best alternative. Because query optimizers are imperfect, database users and administrators sometimes need to manually examine and tune the plans produced by the optimizer to get better performance.

Generating query plans

A given database management system may offer one or more mechanisms for returning the plan for a given query. Some packages feature tools which will generate a graphical representation of a query plan. Other tools allow a special mode to be set on the connection to cause the DBMS to return a textual description of the query plan. Another mechanism for retrieving the query plan involves querying a virtual database table after executing the query to be examined. In Oracle, for instance, this can be achieved using the EXPLAIN PLAN statement.

Graphical plans

The SQL Server Management Studio tool which ships with Microsoft SQL Server, for example, shows this graphical plan when executing this two-table join against a sample database:

```
SELECT *
FROM HumanResources.Employee AS e
    INNER JOIN Person.Contact AS c
    ON e.ContactID = c.ContactID
ORDER BY c.LastName
```

The UI allows exploration of various attributes of the operators involved in the query plan, including the operator type, the number of rows each operator consumes or produces, and the expected cost of each operator's work.

Textual plans

The textual plan given for the same query in the screenshot is shown here:

```
StmtText
----
  |--Sort(ORDER BY:([c].[LastName] ASC))
       |--Nested Loops(Inner Join, OUTER REFERENCES:([e].[ContactID],
[Expr1004]) WITH UNORDERED PREFETCH)
            |--Clustered Index
```

```
Scan(OBJECT:([AdventureWorks].[HumanResources].[Employee].[PK_Employee_EmployeeID]
 AS [e]))
            |--Clustered Index
Seek(OBJECT:([AdventureWorks].[Person].[Contact].[PK_Contact_ContactID]
 AS [c]),

SEEK:([c].[ContactID]=[AdventureWorks].[HumanResources].[Employee].[ContactID]
 as [e].[ContactID]) ORDERED FORWARD)
```

It indicates that the query engine will do a scan over the primary key index on the Employee table and a matching seek through the primary key index (the ContactID column) on the Contact table to find matching rows. The resulting rows from each side will be shown to a nested loops join operator, sorted, then returned as the result set to the connection.

In order to tune the query, the user must understand the different operators that the database may use, and which ones might be more efficient than others while still providing semantically correct query results.

Database tuning

Reviewing the query plan can present opportunities for new indexes or changes to existing indexes. It can also show that the database is not properly taking advantage of existing indexes (see query optimizer).

Query tuning

The query optimizer will not always choose the best query plan for a given query. In some databases the query plan can be reviewed, problems found, and then the query optimizer given hints on how to improve it. In other databases alternatives to express the same query (other queries that return the same results) can be tried. Some query tools can generate embedded hints in the query, for use by the optimizer.

Some databases like Oracle provide a Plan table for query tuning. This plan table will return the cost and time for executing a Query. In Oracle there are 2 optimization techniques:

1. CBO or Cost Based Optimization
2. RBO or Rule Based Optimization

The RBO is slowly being deprecated. For CBO to be used, all the tables referenced by the query must be analyzed. To analyze a table, a package DBMS_STATS can be made use of.

The others methods for query optimization include:

1. SQL Trace
2. Oracle Trace
3. TKPROF

€ Video tutorial on how to perform SQL performance tuning with reference to Oracle [1]

References

[1] http://seeingwithc.org/sqltuning.html

Functions

Database administration and automation

Database administration is the function of managing and maintaining database management systems (DBMS) software. Mainstream DBMS software such as Oracle, IBM DB2 and Microsoft SQL Server need ongoing management. As such, corporations that use DBMS software often hire specialized IT (Information Technology) personnel called Database Administrators or DBAs.

DBA Responsibilities

- Installation, configuration and upgrading of Database server software and related products.
- Evaluate Database features and Database related products.
- Establish and maintain sound backup and recovery policies and procedures.
- Take care of the Database design and implementation.
- Implement and maintain database security (create and maintain users and roles, assign privileges).
- Database tuning and performance monitoring.
- Application tuning and performance monitoring.
- Setup and maintain documentation and standards.
- Plan growth and changes (capacity planning).
- Work as part of a team and provide 24x7 support when required.
- Do general technical troubleshooting and give cons.
- Database recovery.

Types of database administration

There are three types of DBAs:

1. Systems DBAs (also referred to as Physical DBAs, Operations DBAs or Production Support DBAs): focus on the physical aspects of database administration such as DBMS installation, configuration, patching, upgrades, backups, restores, refreshes, performance optimization, maintenance and disaster recovery.
2. Development DBAs: focus on the logical and development aspects of database administration such as data model design and maintenance, DDL (data definition language) generation, SQL writing and tuning, coding stored procedures, collaborating with developers to help choose the most appropriate DBMS feature/functionality and other pre-production activities.
3. Application DBAs: usually found in organizations that have purchased 3rd party application software such as ERP (enterprise resource planning) and CRM (customer relationship management) systems. Examples of such application software includes Oracle Applications, Siebel and PeopleSoft (both now part of Oracle Corp.) and SAP. Application DBAs straddle the fence between the DBMS and the application software and are responsible for ensuring that the application is fully optimized for the database and vice versa. They usually manage all the application components that interact with the database and carry out activities such as application installation and patching, application upgrades, database cloning, building and running data cleanup routines, data load process management, etc.

While individuals usually specialize in one type of database administration, in smaller organizations, it is not uncommon to find a single individual or group performing more than one type of database administration.

Nature of database administration

The degree to which the administration of a database is automated dictates the skills and personnel required to manage databases. On one end of the spectrum, a system with minimal automation will require significant experienced resources to manage; perhaps 5-10 databases per DBA. Alternatively an organization might choose to automate a significant amount of the work that could be done manually therefore reducing the skills required to perform tasks. As automation increases, the personnel needs of the organization splits into highly skilled workers to create and manage the automation and a group of lower skilled "line" DBAs who simply execute the automation.

Database administration work is complex, repetitive, time-consuming and requires significant training. Since databases hold valuable and mission-critical data, companies usually look for candidates with multiple years of experience. Database administration often requires DBAs to put in work during off-hours (for example, for planned after hours downtime, in the event of a database-related outage or if performance has been severely degraded). DBAs are commonly well compensated for the long hours

One key skill required and often overlooked when selecting a DBA is database recovery (under disaster recovery). It is not a case of •if‡ but a case of •when‡ a database suffers a failure, ranging from a simple failure to a full catastrophic failure. The failure may be data corruption, media failure, or user induced errors. In either situation the DBA must have the skills to recover the database to a given point in time to prevent a loss of data. A highly skilled DBA can spend a few minutes or exceedingly long hours to get the database back to the operational point.

Database administration tools

Often, the DBMS software comes with certain tools to help DBAs manage the DBMS. Such tools are called native tools. For example, Microsoft SQL Server comes with SQL Server Enterprise Manager and Oracle has tools such as SQL*Plus and Oracle Enterprise Manager/Grid Control. In addition, 3rd parties such as BMC, Quest Software, Embarcadero Technologies, EMS Database Management Solutions and SQL Maestro Group offer GUI tools to monitor the DBMS and help DBAs carry out certain functions inside the database more easily.

Another kind of database software exists to manage the provisioning of new databases and the management of existing databases and their related resources. The process of creating a new database can consist of hundreds or thousands of unique steps from satisfying prerequisites to configuring backups where each step must be successful before the next can start. A human cannot be expected to complete this procedure in the same exact way time after time - exactly the goal when multiple databases exist. As the number of DBAs grows, without automation the number of unique configurations frequently grows to be costly/difficult to support. All of these complicated procedures can be modeled by the best DBAs into database automation software and executed by the standard DBAs. Software has been created specifically to improve the reliability and repeatability of these procedures such as Stratavia's Data Palette and GridApp Systems Clarity.

The impact of IT automation on database administration

Recently, automation has begun to impact this area significantly. Newer technologies such as Stratavia's Data Palette suite and GridApp Systems Clarity have begun to increase the automation of databases causing the reduction of database related tasks. However at best this only reduces the amount of mundane, repetitive activities and does not eliminate the need for DBAs. The intention of DBA automation is to enable DBAs to focus on more proactive activities around database architecture, deployment, performance and service level management.

Every database requires a database owner account that can perform all schema management operations. This account is specific to the database and cannot log in to Data Director. You can add database owner accounts after database creation. Data Director users must log in with their database-specific credentials to view the database, its entities, and its data or to perform database management tasks. Database administrators and application developers can manage databases only if they have appropriate permissions and roles granted to them by the organization

administrator. The permissions and roles must be granted on the database group or on the database, and they only apply within the organization in which they are granted.

Learning database administration

There are several education institutes that offer professional courses, including late-night programs, to allow candidates to learn database administration. Also, DBMS vendors such as Oracle, Microsoft and IBM offer certification programs to help companies to hire qualified DBA practitioners. College degree in Computer Science or related field is helpful but not necessarily a prerequisite.

External references

- "A set theoretic data structure and retrieval language" [1]. *SIGIR Forum* (ACM Special Interest Group on Information Retrieval) **7** (4): 45€55. Winter 1972.
- Thomas Haigh (June 2006). "Origins of the Data Base Management System" [2] (PDF). *SIGMOD Record* (ACM Special Interest Group on Management of Data) **35** (2).

This article is based on material taken from the Free On-line Dictionary of Computing prior to 1 November 2008 and incorporated under the "relicensing" terms of the GFDL, version 1.3 or later.

References

[1] http://portal.acm.org/citation.cfm?id=1095495.1095500
[2] http://www.tomandmaria.com/tom/Writing/VeritableBucketOfFactsSIGMOD.pdf

Replication (computing)

Replication in computing involves sharing information so as to ensure consistency between redundant resources, such as software or hardware components, to improve reliability, fault-tolerance, or accessibility.

Terminology

One speaks of:

- *data replication* if the same data is stored on multiple storage devices,[1]
- *computation replication* if the same computing task is executed many times.

A computational task is typically *replicated in space*, i.e. executed on separate devices, or it could be *replicated in time*, if it is executed repeatedly on a single device.

The access to a replicated entity is typically uniform with access to a single, non-replicated entity. The replication itself should be transparent to an external user. Also, in a failure scenario, a failover of replicas is hidden as much as possible. The latter refers to data replication with respect to Quality of Service (QoS) aspects.[2]

Computer scientists talk about active and passive replication in systems that replicate data or services:

- *active replication* is performed by processing the same request at every replica.
- *passive replication* involves processing each single request on a single replica and then transferring its resultant state to the other replicas.

If at any time one master replica is designated to process all the requests, then we are talking about the *primary-backup* scheme (*master-slave* scheme) predominant in high-availability clusters. On the other side, if any replica processes a request and then distributes a new state, then this is a *multi-primary* scheme (called *multi-master* in the database field). In the multi-primary scheme, some form of distributed concurrency control must be used, such

as distributed lock manager.

Load balancing differs from task replication, since it distributes a load of different (not the same) computations across machines, and allows a single computation to be dropped in case of failure. Load balancing, however, sometimes uses data replication (especially multi-master replication) internally, to distribute its data among machines.

Backup differs from replication in that it saves a copy of data unchanged for a long period of time.[*citation needed*] Replicas, on the other hand, undergo frequent updates and quickly lose any historical state. Replication is one of the oldest and most important topics in the overall area of distributed systems.

Whether one replicates data or computation, the objective is to have some group of processes that handle incoming events. If we replicate data, these processes are passive and operate only to maintain the stored data, reply to read requests, and apply updates. When we replicate computation, the usual goal is to provide fault-tolerance. For example, a replicated service might be used to control a telephone switch, with the objective of ensuring that even if the primary controller fails, the backup can take over its functions. But the underlying needs are the same in both cases: by ensuring that the replicas see the same events in equivalent orders, they stay in consistent states and hence any replica can respond to queries.

Replication models in distributed systems

A number of widely cited models exist for data replication, each having its own properties and performance:

1. *Transactional replication*. This is the model for replicating transactional data, for example a database or some other form of transactional storage structure. The one-copy serializability model is employed in this case, which defines legal outcomes of a transaction on replicated data in accordance with the overall ACID properties that transactional systems seek to guarantee.
2. *State machine replication.* This model assumes that replicated process is a deterministic finite automaton and that atomic broadcast of every event is possible. It is based on a distributed computing problem called *distributed consensus* and has a great deal in common with the transactional replication model. This is sometimes mistakenly used as synonym of *active replication*. State machine replication is usually implemented by a replicated log consisting of multiple subsequent rounds of the Paxos algorithm. This was popularized by Google's Chubby system, and is the core behind the open-source Keyspace data store.
3. *Virtual synchrony*. This computational model is used when a group of processes cooperate to replicate in-memory data or to coordinate actions. The model defines a distributed entity called a *process group*. A process can join a group, and is provided with a checkpoint containing the current state of the data replicated by group members. Processes can then send multicasts to the group and will see incoming multicasts in the identical order. Membership changes are handled as a special multicast that delivers a new *membership view* to the processes in the group.

Database replication

Database replication can be used on many database management systems, usually with a master/slave relationship between the original and the copies. The master logs the updates, which then ripple through to the slaves. The slave outputs a message stating that it has received the update successfully, thus allowing the sending (and potentially re-sending until successfully applied) of subsequent updates.

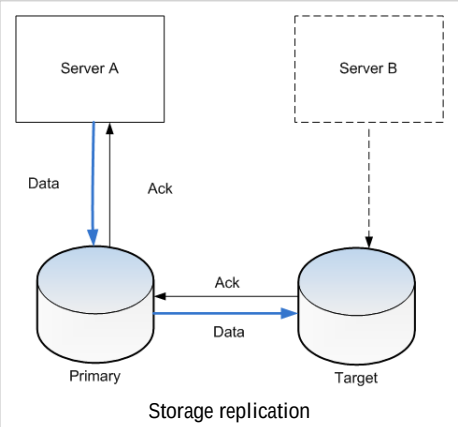
Multi-master replication, where updates can be submitted to any database node, and then ripple through to other servers, is often desired, but introduces substantially increased costs and complexity which may make it impractical in some situations. The most common challenge that exists in multi-master replication is transactional conflict prevention or resolution. Most synchronous or eager replication solutions do conflict prevention, while asynchronous solutions have to do conflict resolution. For instance, if a record is changed on two nodes simultaneously, an eager

replication system would detect the conflict before confirming the commit and abort one of the transactions. A lazy replication system would allow both transactions to commit and run a conflict resolution during resynchronization. The resolution of such a conflict may be based on a timestamp of the transaction, on the hierarchy of the origin nodes or on much more complex logic, which decides consistently on all nodes.

Database replication becomes difficult when it scales up. Usually, the scale up goes with two dimensions, horizontal and vertical: horizontal scale-up has more data replicas, vertical scale-up has data replicas located further away in distance. Problems raised by horizontal scale-up can be alleviated by a multi-layer multi-view access protocol. Vertical scale-up causes fewer problems in that internet reliability and performance are improving.

When data is replicated between database servers, so that the information remains consistent throughout the database system and users cannot tell or even know which server in the DBMS they are using, the system is said to exhibit replication transparency.

Disk storage replication



Storage replication

Active (real-time) storage replication is usually implemented by distributing updates of a block device to several physical hard disks. This way, any file system supported by the operating system can be replicated without modification, as the file system code works on a level above the block device driver layer. It is implemented either in hardware (in a disk array controller) or in software (in a device driver).

The most basic method is disk mirroring, typical for locally-connected disks. The storage industry narrows the definitions, so *mirroring* is a local (short-distance) operation. A *replication* is extendable across a computer network, so the disks can be located in physically distant locations, and the master-slave database replication model is usually applied. The purpose of replication is to prevent damage from failures or disasters that may occur in one location, or in case such events do occur, improve the ability to recover. For replication, latency is the key factor because it determines either how far apart the sites can be or the type of replication that can be employed.

The main characteristic of such cross-site replication is how write operations are handled:

- Synchronous replication - guarantees "zero data loss" by the means of atomic write operation, i.e. write either completes on both sides or not at all. Write is not considered complete until acknowledgement by both local and remote storage. Most applications wait for a write transaction to complete before proceeding with further work, hence overall performance decreases considerably. Inherently, performance drops proportionally to distance, as latency is caused by speed of light. For 10ƒkm distance, the fastest possible roundtrip takes 67 “s, whereas nowadays a whole local cached write completes in about 10-20 “s.
  - An often-overlooked aspect of synchronous replication is the fact that failure of *remote* replica, or even just the *interconnection*, stops by definition any and all writes (freezing the local storage system). This is the behaviour that guarantees zero data loss. However, many commercial systems at such potentially dangerous point do not freeze, but just proceed with local writes, losing the desired zero recovery point objective.
  - The main difference between synchronous and asynchronous volume replication is that synchronous replication needs to wait for the destination server in any write operation.[3]
- Asynchronous replication - write is considered complete as soon as local storage acknowledges it. Remote storage is updated, but probably with a small lag. Performance is greatly increased, but in case of losing a local storage, the remote storage is not guaranteed to have the current copy of data and most recent data may be lost.

- Semi-synchronous replication - this usually means[*citation needed*] that a write is considered complete as soon as local storage acknowledges it and a remote server acknowledges that it has received the write either into memory or to a dedicated log file. The actual remote write is not performed immediately but is performed asynchronously, resulting in better performance than synchronous replication but offering no guarantee of durability.
  - Point-in-time replication - introduces periodic snapshots that are replicated instead of primary storage. If the replicated snapshots are pointer-based, then during replication only the changed data is moved not the entire volume. Using this method, replication can occur over smaller, less expensive bandwidth links such as iSCSI or T1 instead of fiber optic lines.

To address the limits imposed by latency, techniques of WAN optimization can be applied to the link.

Notable implementations

Many distributed filesystems use replication to ensure fault tolerance and avoid a single point of failure. See the lists of distributed fault-tolerant file systems and distributed parallel fault-tolerant file systems.

Other notable storage replication software includes:

- CA - ARCserve [4] Replication and High Availability RHA [5]
- Dell - AppAssure Backup, replication and disaster recovery
- Dell - Compellent Remote Instant Replay
- EMC - EMC RecoverPoint
- EMC - EMC SRDF
- EMC - EMC VPLEX
- DataCore SANsymphony & SANmelody
- StarWind iSCSI SAN & NAS
- FalconStor Replication & Mirroring (sub-block heterogeneous point-in-time, async, sync)
- FreeNas - Replication handled by ssh + zfs file system [6]
- Hitachi TrueCopy
- Hewlett-Packard - Continuous Access (HP CA)
- IBM - Peer to Peer Remote Copy (PPRC) and Global Mirror (known together as IBM Copy Services)
- Linux - DRBD - open source module
- HAST DRBD-like Open Source solution for FreeBSD.
- MapR volume mirroring
- NetApp SyncMirror
- NetApp SnapMirror
- Symantec Veritas Volume Replicator (VVR)
- VMware - Site Recovery Manager (SRM) [7]

File-based replication

File-based replication is replicating files at a logical level rather than replicating at the storage block level. There are many different ways of performing this. Unlike with storage-level replication, the solutions almost exclusively rely on software.

Capture with a kernel driver

With the use of a kernel driver (specifically a filter driver), that intercepts calls to the filesystem functions, any activity is captured immediately as it occurs. This utilises the same type of technology that real time active virus checkers employ. At this level, logical file operations are captured like file open, write, delete, etc. The kernel driver transmits these commands to another process, generally over a network to a different machine, which will mimic the

operations of the source machine. Like block-level storage replication, the file-level replication allows both synchronous and asynchronous modes. In synchronous mode, write operations on the source machine are held and not allowed to occur until the destination machine has acknowledged the successful replication. Synchronous mode is less common with file replication products although a few solutions exists.[8]

File level replication solution yield a few benefits. Firstly because data is captured at a file level it can make an informed decision on whether to replicate based on the location of the file and the type of file. Hence unlike block-level storage replication where a whole volume needs to be replicated, file replication products have the ability to exclude temporary files or parts of a filesystem that hold no business value. This can substantially reduce the amount of data sent from the source machine as well as decrease the storage burden on the destination machine. A further benefit to decreasing bandwidth is the data transmitted can be more granular than with block-level replication. If an application writes 100 bytes, only the 100 bytes are transmitted not a complete disk block which is generally 4096 bytes.

On a negative side, as this is a software only solution, it requires implementation and maintenance on the operating system level, and uses some of machine's processing power (CPU).

Notable implementations:

- CA ARCserve [4] Replication [5]
- Cofio Software AIMstor Replication [9]
- Double-Take Software Availability [10]
- EDpCloud Software EDpCloud Real Time Replication [11]

Filesystem journal replication

In many ways working like a database journal, many filesystems have the ability to journal their activity. The journal can be sent to another machine, either periodically or in real time. It can be used there to play back events.

Notable implementations:

- Microsoft DPM (periodical updates, not in real time)

Batch replication

This is the process of comparing the source and destination filesystems and ensuring that the destination matches the source. The key benefit is that such solutions are generally free or inexpensive. The downside is that the process of synchronizing them is quite system-intensive, and consequently this process generally runs infrequently.

Notable implementations:

- rsync

Distributed shared memory replication

Another example of using replication appears in distributed shared memory systems, where it may happen that many nodes of the system share the same page of the memory - which usually means, that each node has a separate copy (replica) of this page.

Primary-backup and multi-primary replication

Many classical approaches to replication are based on a primary/backup model where one device or process has unilateral control over one or more other processes or devices. For example, the primary might perform some computation, streaming a log of updates to a backup (standby) process, which can then take over if the primary fails. This approach is the most common one for replicating databases, despite the risk that if a portion of the log is lost during a failure, the backup might not be in a state identical to the one the primary was in, and transactions could

then be lost.

A weakness of primary/backup schemes is that in settings where both processes could have been active, only one is actually performing operations. We're gaining fault-tolerance but spending twice as much money to get this property. For this reason, starting in the period around 1985, the distributed systems research community began to explore alternative methods of replicating data. An outgrowth of this work was the emergence of schemes in which a group of replicas could cooperate, with each process backup up the others, and each handling some share of the workload.

Jim Gray, a towering figure[12] within the database community, analyzed multi-primary replication schemes under the transactional model and ultimately published a widely cited paper skeptical of the approach "The Dangers of Replication and a Solution [13]". In a nutshell, he argued that unless data splits in some natural way so that the database can be treated as *n* disjoint sub-databases, concurrency control conflicts will result in seriously degraded performance and the group of replicas will probably slow down as a function of *n*. Indeed, he suggests that the most common approaches are likely to result in degradation that scales as *O(n,)*. His solution, which is to partition the data, is only viable in situations where data actually has a natural partitioning key.

The situation is not always so bleak. For example, in the 1985-1987 period, the virtual synchrony model was proposed and emerged as a widely adopted standard (it was used in the Isis Toolkit, Horus, Transis, Ensemble, Totem, Spread, C-Ensemble, Phoenix and Quicksilver systems, and is the basis for the CORBA fault-tolerant computing standard; the model is also used in IBM Websphere to replicate business logic and in Microsoft's Windows Server 2008 enterprise clustering technology). Virtual synchrony permits a multi-primary approach in which a group of processes cooperate to parallelize some aspects of request processing. The scheme can only be used for some forms of in-memory data, but when feasible, provides linear speedups in the size of the group.

A number of modern products support similar schemes. For example, the Spread Toolkit supports this same virtual synchrony model and can be used to implement a multi-primary replication scheme; it would also be possible to use C-Ensemble or Quicksilver in this manner. WANdisco permits active replication where every node on a network is an exact copy or replica and hence every node on the network is active at one time; this scheme is optimized for use in a wide area network.

References

[1] http://searchsqlserver.techtarget.com/definition/database-replication

[2] V. Andronikou, K. Mamouras, K. Tserpes, D. Kyriazis, T. Varvarigou, *Dynamic QoS-aware Data Replication in Grid Environments*, Elsevier Future Generation Computer Systems - The International Journal of Grid Computing and eScience, 2012

[3] Open-E Knowledgebase. "What is the difference between asynchronous and synchronous volume replication?" (http://kb.open-e.com/What-is-the-difference-between-asynchronous-and-synchronous-volume-replication-_682.html) 12 August 2009.

[4] http://www.arcserve.com/gb/default.aspx

[5] http://www.arcserve.com/gb/products/ca-arcserve-replication/ca-arcserve-replication-features-overview.aspx

[6] http://doc.freenas.org/index.php/Replication_Tasks

[7] http://pubs.vmware.com/srm-51/index.jsp?topic=%2Fcom.vmware.srm.install_config.doc%2FGUID-B3A49FFF-E3B9-45E3-AD35-093D896596A0.html

[8] AIMstor Replication (http://www.cofio.com/AIMstor-Replication/)

[9] http://www.cofio.com/AIMstor-Replication/

[10] http://www.doubletake.com/uk/products/double-take-availability/Pages/default.aspx

[11] http://www.enduradata.com/

[12] *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data: SIGMOD '99*, Philadelphia, PA, USA; June 1€3, 1999, Volume 28; p. 3.

[13] http://research.microsoft.com/~gray/replicas.ps

Database Products

Comparison of object database management systems

This is a **comparison of notable object database management systems**, showing what fundamental object database features are implemented natively.

Name	Current Stable Version	Language(s)	SQL support	Datatypes	License	Description
Cach„	2012.1	ObjectScript (dynamic language), Basic. Java/.NET object mapping supported.	SQL subset. Object notation allowed. Supports embedded SQL, dynamic SQL and xDBC access.		Proprietary	MUMPS ancestry. Includes built-in support for XML, Web/AJAX and an EMB system called Ensemble. Supports embedded, client/server and distributed implementations.
ConceptBase		Telos	CBQL (based on Datalog)	no types but classes	open source, FreeBSD-style license	historical db, active rules, meta-modeling, deductive rules
Db4o	8.0	C#, Java	db4o-sql [1]	.NET and Java data types	GPL, custom,[2] proprietary	Native Queries, LINQ support, automatic schema evolution, Transparent Activation/Persistence, replication to RDBMS, Object Manager plugin for Visual Studio and Eclipse
Gemstone		Smalltalk, Java				
NeoDatis ODB		C#, Java, Mono			LGPL	Embedded and Client/Server
ObjectDatabase++	3.4	C++, TScript, .NET			Proprietary	Embedded
ObjectDB	2.4.6	Java	None, uses JPA or JDO		Proprietary	
Objectivity/DB	10.2.1	C++, C#, Java, Python, Smalltalk and XML	SQL superset		Proprietary	Distributed, Parallel Query Engine
ObjectStore	7.2 (July 2011)	C++, Java, interoperable with .NET	SQL subset (also has own object query language)		Proprietary	Embedded database supporting efficient, distributed management of C++ and Java objects. Avoids the complexities and limitations of ORM products such as Hibernate by storing objects directly with their relationships intact. Uses a page-based mapping system for fast locking and efficient, distributed, client-side caching.
ODABA		C++, .NET			GPL	Terminology-oriented database
OpenAccess	2.2	C++	no		Proprietary	EDA database

OpenLink Virtuoso	5.0.11	C++, Java/JSP, ASP, ASPX, Mono, RDF, SPARQL, SPARUL, SQL, Perl, Python, PHP, Ruby, XML, ODBC, JDBC, ADO.NET, more	SQL 9x/200x		GPL or proprietary	
Perst	4.2	Java (including Java SE, Java ME & Android), C# (including .NET, .NET Compact Framework, Mono & Silverlight)	JSQL - object-oriented subset of SQL	Java and .NET data types	GPL, Proprietary	Small footprint embedded database. Diverse indexes and specialized collection classes; LINQ; replication; ACID transactions; native full text search; includes Silverlight, Android and Java ME demo apps.
Picolisp	3.1.1	Picolisp			MIT License	DB built into the language
Twig		Java			Apache license 2.0	Built on Google App Engine's low-level Datastore API
Versant Object Database					Proprietary	
WakandaDB	4	JavaScript, C++	No support. Use REST & SSJS instead	JavaScript and 4D data types	AGPL, proprietary[3]	NoSQL REST / Server-Side JavaScript engine. Integrates Webkit JavaScriptCore engine with HTML5 JS APIs supported on the server. Tables and columns are replaced by JavaScript DataClasses and attributes.
Zope Object Database		Python, C	No support. Object indexing and searching is done through ZCatalog facility.		Zope Public License	

References

[1] http://code.google.com/p/db4o-sql/
[2] http://www.db4o.com/about/company/legalpolicies/docl.aspx
[3] Wakanda Commercial license (http://www.wakanda.org/license/commercial)

Comparison of object-relational database management systems

This is a **comparison of object-relational database management systems** (ORDBMSs). Each system has at least some features of an object-relational database; they vary widely in their completeness and the approaches taken.

The following tables compare general and technical information; please see the individual products' articles for further information. This article is not all-inclusive nor necessarily up to date. Unless otherwise specified in footnotes, comparisons are based on the stable versions without any add-ons, extensions or external programs.

Basic data

Name	Vendor	License	OS	Notes
Adaptive Server Enterprise	SAP	Proprietary	Cross-platform	
CUBRID	NHN Corporation	GPL/BSD	Linux, Windows	
DB2	IBM	Proprietary	Cross-platform	
GigaBASE	knizhnik	MIT	Various	SourceForge download page [1]
Greenplum Database	Greenplum division of EMC Corporation	Proprietary	?	Uses PostgreSQL codebase
Informix	IBM	Proprietary	Cross-platform	
Cach…	InterSystems	Proprietary		
LogicSQL	Shanghai Shifang Software, Inc.	unknown license	Download page [2]	
Microsoft SQL Server	Microsoft Corporation	Proprietary	Windows	Supports data objects in .NET languages
Oracle Database	Oracle Corporation	Proprietary	Linux, Windows, Unix	
PostgreSQL	PostgreSQL Global Development Group	Postgres	Cross-platform	
OpenEdge Advanced Business Language (formerly Progress 4GL)	Progress Software Corporation	Proprietary	Cross-platform	
Valentina	Paradigma Software	Proprietary	Windows, Linux, Mac OS X	Web site [3]
Virtuoso Universal Server	OpenLink Software	GPLv2 or proprietary	Cross-platform	
VMDS (Version Managed Data Store)	GE Energy, a division of General Electric	Proprietary	?	GIS for public utilities; can be stored inside Oracle Database
WakandaDB	4D	AGPLv3 or proprietary	Windows, Linux, Mac OS X	Based on REST and Server-Side JavaScript
Zope Object Database	Zope Corporation	Zope Public License	Cross-platform	For Python, also included in Zope web application server

Object features

Information about what fundamental ORDBMSes features are implemented natively.

	Type	Method	Type inheritance	Table inheritance
CUBRID	Yes	Yes	Yes	Yes
LogicSQL[4]	*?*	*?*	*?*	*?*
Oracle	Yes	Yes[5]	Yes	Yes
OpenLink Virtuoso	Yes	Yes	Yes	Yes
PostgreSQL	Yes	Yes	Yes	Yes
Informix	Yes	Yes	Yes	Yes
WakandaDB	Yes	Yes	Yes	Yes

Data types

Information about what data types are implemented natively.

	Array	List	Set	Multiset	Object reference
CUBRID	Yes	Yes	Yes	Yes	Yes
LogicSQL	*?*	*?*	*?*	*?*	*?*
Oracle	Yes	Yes	Yes	Yes	Yes
OpenLink Virtuoso	Yes	Yes	Yes	Yes	Yes
PostgreSQL	Yes	Yes	Yes	Yes	Yes
Informix	No	Yes	Yes	Yes	Yes

References

[1] http://sourceforge.net/projects/gigabase/
[2] http://webdocs.cs.ualberta.ca/~yuan/databases/index.html
[3] http://www.valentina-db.com/
[4] http://webdocs.cs.ualberta.ca/~yuan/databases/index.html
[5] No private methods, no way to call super method from a child.

External links

€ Arvin.dk (http://troels.arvin.dk/db/rdbms/), Comparison of different SQL implementations

List of relational database management systems

This is a **list of relational database management systems**.

List of Software

- 4th Dimension
- Adabas D
- Alpha Five
- Apache Cassandra
- Apache Derby
- Aster Data
- Altibase
- BlackRay
- CA-Datacom
- Clarion
- Clustrix
- CSQL
- CUBRID
- Daffodil database
- DataEase
- Database Management Library
- Dataphor
- dBase
- Derby aka Java DB
- Empress Embedded Database
- EXASolution
- EnterpriseDB
- eXtremeDB
- FileMaker Pro
- Firebird
- Greenplum
- GroveSite
- H2
- Helix database
- HSQLDB
- IBM DB2
- IBM Lotus Approach
- IBM DB2 Express-C
- Infobright
- Informix
- Ingres
- InterBase
- InterSystems Cach…
- GT.M
- Linter
- MariaDB

- MaxDB
- MemSQL
- Microsoft Access
- Microsoft Jet Database Engine (part of Microsoft Access)
- Microsoft SQL Server
- Microsoft SQL Server Express
- Microsoft Visual FoxPro
- Mimer SQL
- MonetDB
- mSQL
- MySQL
- Netezza
- NexusDB
- NonStop SQL
- Openbase
- OpenLink Virtuoso (Open Source Edition)
- OpenLink Virtuoso Universal Server
- OpenOffice.org Base
- Oracle
- Oracle Rdb for OpenVMS
- Panorama
- Pervasive PSQL
- Polyhedra
- PostgreSQL
- Postgres Plus Advanced Server
- Progress Software
- RDM Embedded
- RDM Server
- The SAS system
- SAND CDBMS
- SAP HANA
- SAP Sybase Adaptive Server Enterprise
- SAP Sybase IQ
- SQL Anywhere (formerly known as Sybase Adaptive Server Anywhere and Watcom SQL)
- ScimoreDB
- SmallSQL
- solidDB
- SQLBase
- SQLite
- Sybase Advantage Database Server
- Teradata
- TimesTen
- txtSQL
- mizanSQL
- Unisys RDMS 2200
- UniData
- UniVerse

- Vertica
- VMDS

Historical

- Britton Lee IDMs
- Cornerstone
- IBM System R
- MICRO Information Management System
- Oracle Rdb
- Paradox
- Pick
- PRTV
- QBE
- IBM SQL/DS
- Sybase SQL Server

Relational by the Date-Darwen-Pascal Model

Current

- Alphora Dataphor (a proprietary virtual, federated DBMS and RAD MS .Net IDE).
- Rel (free Java implementation).

Obsolete

- IBM Business System 12
- IBM IS1
- IBM PRTV (ISBL)
- Multics Relational Data Store

Comparison of relational database management systems

The following tables compare general and technical information for a number of relational database management systems. Please see the individual products' articles for further information. This article is not all-inclusive or necessarily up to date. Unless otherwise specified in footnotes, comparisons are based on the stable versions without any add-ons, extensions or external programs.

General information

	Maintainer	First public release date	Latest stable version	Latest release date	Software license
4D (4th Dimension)	4D S.A.S.	1984	v13.2	2012-11-12	Proprietary
ADABAS	Software AG	1970	8.1	2013-06	Proprietary
Adaptive Server Enterprise	Sybase	1987	15.7		Proprietary
Advantage Database Server (ADS)	Sybase	1992	11.1	2012	Proprietary
Altibase	Altibase Corp.	2000	6.1.1	2012-04-01	Proprietary
Apache Derby	Apache	2004	10.10.1.1	2013-04-15	Apache License
Clustrix	Clustrix	2010	v5.0	2013-05-01	Proprietary
CUBRID	NHN Corporation	2008-11	8.4.1	2012-02-24	GPL v2
Datacom	CA, Inc.	***?***	11.2		Proprietary
DB2	IBM	1983	10.5	2013-04-23	Proprietary
Drizzle	Brian Aker	2008	7.1.36	2012-05-23	BSD, GPL v2
Empress Embedded Database	Empress Software Inc	1979	10.20	2010-03	Proprietary
EXASolution	EXASOL AG	2004	4.1	2012-07-17	Proprietary
Firebird	Firebird project	2000-07-25	2.5.2	2013-03-24	IPL and IDPL
HSQLDB	HSQL Development Group	2001	2.2.9[1]	2013-07-08	BSD
H2	H2 Software	2005	1.3.171	2013-03-17	EPL and modified MPL
Informix Dynamic Server	IBM	1980	12.10.xC1	2013-03-26	Proprietary
Ingres	Ingres Corp.	1974	Ingres Database 10	2010-10-12	GPL and Proprietary
InterBase	Embarcadero	1984	InterBase XE	2010-09-21	Proprietary
Linter SQL RDBMS	RELEX Group	1990	6.x	2013-08-26	Proprietary
LucidDB	The Eigenbase Project	2007-01	0.9.3		GPL v2
MariaDB	MariaDB Community	2010-02-01	5.5.30	2013-03-12	GPL v2
MaxDB	SAP AG	2003-05	7.6	2008-01	Proprietary
Microsoft Access (JET)	Microsoft	1992	15 (2013)	2012-10-02	Proprietary
Microsoft Visual Foxpro	Microsoft	1984	9 (2005)	2007-10-11	Proprietary
Microsoft SQL Server	Microsoft	1989	2012 (v11)		Proprietary

Microsoft SQL Server Compact (Embedded Database)	Microsoft	2000	2011 (v4.0)		Proprietary
MonetDB/SQL	The MonetDB Developer Team	2004	11.9.1	2012-04	MonetDB Public License v1.1
mSQL	Hughes Technologies	1994	3.9	2011-02	Proprietary
MySQL	Sun Microsystems (now Oracle Corporation)	1995-11	5.6.31	2013-07-30	GPL or Proprietary
MemSQL	MemSQL	2012-06	1.8 (2012)	2012-12	Proprietary
Nexusdb	Nexus Database Systems Pty Ltd	2003-09	3.04	2010-05-08	Proprietary
HP NonStop SQL	Hewlett-Packard	1987	SQL/MX 2.3		Proprietary
Omnis Studio	TigerLogic Inc	1982-07	4.3.1 Release 1no	2008-05	Proprietary
OpenBase SQL	OpenBase International	1991	11.0.0		Proprietary
OpenEdge	Progress Software Corporation	1984	11.0		Proprietary
OpenLink Virtuoso	OpenLink Software	1998	7.x	2013-08-05	GPL or Proprietary
Oracle	Oracle Corporation	1979-11	12c Release 1	2013-06-25	Proprietary
Oracle Rdb	Oracle Corporation	1984	7.2.5.3.0	2013-07-16	Proprietary
Paradox	Corel Corporation	1985	11	2003	Proprietary
Pervasive PSQL	Pervasive Software	1982	v11 SP3	2013	Proprietary
Polyhedra DBMS	ENEA AB	1993	8.7	2013-03	Proprietary
PostgreSQL	PostgreSQL Global Development Group	1989-06	9.3.1	2013-10-10	PostgreSQL Licence (a liberal Open Source license)
R:Base	R:BASE Technologies	1982	9.5		Proprietary
RDM	Raima Inc.	1984	11.0	2012-06-29	Proprietary
RDM Server	Raima Inc.	1993	8.4	2012-10-31	Proprietary
SAP HANA	SAP AG	2010	1.0		Proprietary
ScimoreDB	Scimore	2005	3.0	2008-03-03	Proprietary
SmallSQL	SmallSQL	2005-04-16	0.20	2008-12	LGPL
SQL Anywhere	Sybase	1992	12.0	2010-07-09	Proprietary
SQLBase	Unify Corp.	1982	11.5	2008-11	Proprietary
SQLite	D. Richard Hipp	2000-08-17	3.8.0.2	2013-09-03	Public domain
Superbase	Superbase	1984	Scientific (2004)		Proprietary
Teradata	Teradata	1984	14.10		Proprietary
UniData	Rocket Software	1988	7.2.12	2011-10	Proprietary
Xeround Cloud Database	Xeround Systems	2010	3.1	2011-10-11	SaaS

Operating system support

The operating systems that the RDBMSes can run on.

	Windows	OS X	Linux	BSD	UNIX	AmigaOS	Symbian	z/OS	iOS	Android
4th Dimension	Yes	Yes	No	No	No	No	No	No	No	No
ADABAS	Yes	No	Yes	No	Yes	No	No	Yes	No	No
Adaptive Server Enterprise	Yes	No	Yes	Yes	Yes	No	No	No	Yes	Yes
Advantage Database Server	Yes	No	Yes	No	No	No	No	No	No	No
Altibase	Yes	No	Yes	No	Yes	No	No	No	No	No
Apache Derby	Yes	Yes	Yes	Yes	Yes	No	No	Yes	?	No
Clustrix	No	No	Yes	No	Yes	No	No	No	No	No
CUBRID	Yes	Partial	Yes	No	No	No	No	No	No	No
Drizzle	No	Yes	Yes	Yes	Yes	No	No	No	No	No
DB2	Yes	Yes (Express C)	Yes	No	Yes	No	No	Yes	Yes	No
Empress Embedded Database	Yes	Yes	Yes	Yes	Yes	No	No	No	No	Yes
EXASolution	No	No	Yes	No	No	No	No	No	No	No
Firebird	Yes	Yes	Yes	Yes	Yes	No	No	Maybe	No	No
HSQLDB	Yes	Yes	Yes	Yes	Yes	No	No	Yes	?	?
H2	Yes	Yes	Yes	Yes	Yes	No	No	Yes	?	Yes
FileMaker	Yes	Yes	No	No	No	No	No	No	Yes	No
Informix Dynamic Server	Yes	Yes	Yes	Yes	Yes	No	No	Yes	No	No
Ingres	Yes	Yes	Yes	Yes	Yes	No	No	Partial	No	No
InterBase	Yes	Yes	Yes	No	Yes (Solaris)	No	No	No	No	No
Linter SQL RDBMS	Yes	Yes	Yes	Yes	Yes	No	No	Under Linux on System z	No	Yes
LucidDB	Yes	Yes	Yes	No	No	No	No	No	No	No
MariaDB	Yes	Yes	Yes	Yes	Yes	No	No	No	?	?
MaxDB	Yes	No	Yes	No	Yes	No	No	Maybe	No	No
Microsoft Access (JET)	Yes	No	No	No	No	No	No	No	No	No
Microsoft Visual Foxpro	Yes	No	No	No	No	No	No	No	No	No
Microsoft SQL Server	Yes	No	No	No	No	No	No	No	No	No
Microsoft SQL Server Compact (Embedded Database)	Yes	No	No	No	No	No	No	No	No	No
MonetDB/SQL	Yes	Yes	Yes	No	Yes	No	No	No	?	?
MySQL	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	?	Yes[2]
Omnis Studio	Yes	Yes	Yes	No	No	No	No	No	No	No
OpenBase SQL	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
OpenEdge	Yes	No	Yes	No	Yes	No	No	No	No	No
OpenLink Virtuoso	Yes	Yes	Yes	Yes	Yes	No	No	Yes	No	No

Oracle	Yes	Yes	Yes	No	Yes	No	No	Yes	No	No
Oracle Rdb	No	No	No	No	No	No	No	No	No	No
Pervasive PSQL	Yes	Yes (OEM only)	Yes	No	No	No	No	No	No	No
Polyhedra	Yes	No	Yes	No	Yes	No	No	No	No	No
PostgreSQL	Yes	Yes	Yes	Yes	Yes	No	No	Under Linux on System z[3]	No	Yes
R:Base	Yes	No	No	No	No	No	No	No	No	No
RDM	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes	No
RDM Server	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
ScimoreDB	Yes	No	No	No	No	No	No	No	No	No
SmallSQL	Yes	Yes	Yes	Yes	Yes	No	No	Yes	No	No
SQL Anywhere	Yes	Yes	Yes	No	Yes	No	No	No	No	Yes
SQLBase	Yes	No	Yes	No	No	No	No	No	No	No
SQLite	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Maybe	Yes	Yes
Superbase	Yes	No	No	No	No	Yes	No	No	No	No
Teradata	Yes	No	Yes	No	Yes	No	No	No	No	No
UniData	Yes	No	Yes	No	Yes	No	No	No	No	No
UniVerse	Yes	No	Yes	No	Yes	No	No	No	No	No
Xeround Cloud Database	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Fundamental features

Information about what fundamental RDBMS features are implemented natively.

	ACID	Referential integrity	Transactions	Unicode	Interface
4th Dimension	Yes	Yes	Yes	Yes	GUI & SQL
ADABAS	Yes	No	Yes	Yes	proprietary direct call & SQL (via 3rd party)
Adaptive Server Enterprise	Yes	Yes	Yes	Yes	SQL
Advantage Database Server	Yes	Yes	Yes	Yes[4]	API & SQL
Altibase	Yes	Yes	Yes	Yes	API & GUI & SQL
Apache Derby	Yes	Yes	Yes	Yes	SQL
Clustrix	Yes	Yes	Yes	Yes	SQL
CUBRID	Yes	Yes	Yes	Yes	GUI & SQL
Drizzle	Yes	Yes	Yes	Yes	SQL
DB2	Yes	Yes	Yes	Yes	GUI & SQL
Empress Embedded Database	Yes	Yes	Yes	Yes	API & SQL
EXASolution	Yes	Yes	Yes	Yes	API & GUI & SQL
Firebird	Yes	Yes	Yes	Yes	SQL

HSQLDB	Yes	Yes	Yes	Yes	SQL
H2	Yes	Yes	Yes	Yes	SQL
Informix Dynamic Server	Yes	Yes	Yes	Yes	SQL
Ingres	Yes	Yes	Yes	Yes	SQL & QUEL
InterBase	Yes	Yes	Yes	Yes	SQL
Linter SQL RDBMS	Yes	Yes	Yes	Yes	GUI & SQL
LucidDB	Yes	No	No	Yes	SQL
MariaDB	Yes[2]	Partial[3]	Yes[2] except for DDL[4]	Yes	SQL
MaxDB	Yes	Yes	Yes	Yes	SQL
Microsoft Access (JET)	Yes	Yes	Yes	Yes	GUI & SQL
Microsoft Visual FoxPro	No	Yes	Yes	No	GUI & SQL
Microsoft SQL Server	Yes	Yes	Yes	Yes	GUI & SQL
Microsoft SQL Server Compact (Embedded Database)	Yes	Yes	Yes	Yes	GUI & SQL
MonetDB/SQL	Yes	Yes	Yes	Yes	SQL
MySQL	Yes[2]	Partial[3]	Yes[2] except for DDL	Yes	GUI[5] & SQL
OpenBase SQL	Yes	Yes	Yes	Yes	GUI & SQL
Oracle	Yes	Yes	Yes except for DDL	Yes	API & GUI & SQL
Oracle Rdb	Yes	Yes	Yes	Yes	SQL
OpenLink Virtuoso	Yes	Yes	Yes	Yes	API & GUI & SQL
Pervasive PSQL	Yes	Yes	Yes	Yes[6]	API & GUI & SQL
Polyhedra DBMS	Yes	Yes	Yes	Yes	API & SQL
PostgreSQL	Yes	Yes	Yes	Yes	API & GUI & SQL
RDM	Yes	Yes	Yes	Yes	SQL & API
RDM Server	Yes	Yes	Yes	Yes	SQL & API
ScimoreDB	Yes	Yes	Yes	Partial	SQL
SQL Anywhere	Yes	Yes	Yes	Yes	SQL
SQLBase	Yes	Yes	Yes	Yes	API & GUI & SQL
SQLite	Yes	Yes	Yes	Optional[5]	API & SQL
Teradata	Yes	Yes	Yes	Yes	SQL
UniData	Yes	No	Yes	Yes	Multiple
UniVerse	Yes	No	Yes	Yes	Multiple
Xeround Cloud Database	Yes	No	Yes	Yes	SQL
	ACID	**Referential integrity**	**Transactions**	**Unicode**	**Interface**

Note (1): Currently only supports read uncommited transaction isolation. Version 1.9 adds serializable isolation and version 2.0 will be fully ACID compliant.

Note (2): MySQL provides ACID compliance through the default InnoDB storage engine.

Note (3): "For other [than InnoDB] storage engines, MySQL Server parses and ignores the FOREIGN KEY and REFERENCES syntax in CREATE TABLE statements. The CHECK clause is parsed but ignored by all storage engines."

Note (4): Support for Unicode is new in version 10.0.

Note (5): MySQL provides GUI interface through MySQL Workbench.

Note (6): Pervasive PSQL provides UTF-8 storage.

Limits

Information about data size limits.

	Max DB size	Max table size	Max row size	Max columns per row	Max Blob/Clob size	Max CHAR size	Max NUMBER size	Min DATE value	Max DATE value	Max column name size
4th Dimension	Limited	?	?	65,135	200 GB (2 GiB Unicode)	200 GB (2 GiB Unicode)	64 bits	?	?	?
Advantage Database Server	Unlimited	16 EiB	65,530 B	65,135 / (10+ AvgFieldNameLength)	4 GiB	?	64 bits	?	?	128
Apache Derby	Unlimited	Unlimited	Unlimited	1,012 (5,000 in views)	2,147,483,647 chars	254 (VARCHAR: 32,672)	64 bits	0001-01-01	9999-12-31	128
Clustrix	Unlimited	Unlimited	64 MB on Appliance, 4 MB on AWS	?	64 MB	64 MB	64 MB	0001-01-01	9999-12-31	254
CUBRID	2 EB	2 EB	Unlimited	6400	Unlimited	1 GB	64 bits	0001-01-01	9999-12-31	254
Drizzle	Unlimited	64 TB	8 KB	1,000	4 GB (longtext, longblob)	64 KB (text)	64 bits	0001	9999	64
DB2	Unlimited	2 ZB	32,677 B	1,012	2 GB	32 KiB)	64 bits	0001-01-01	9999-12-31	128
Empress Embedded Database	Unlimited	2^{63}-1 bytes	2 GB	32,767	2 GB	2 GB	64 bits	0000-01-01	9999-12-31	32
EXASolution	Unlimited	Unlimited	Unlimited	10,000	N/A	2 MB	128 bits	0001-01-01	9999-12-31	256
FileMaker	8 TB	8 TB	8 TB	256,000,000	4 GB	10^9 characters	10^9 numbers w/ range 10^-400 to 10^400	0001-01-01	4000-12-31	100
Firebird	Unlimited[1]	~32 TB	65,536 B	Depends on data types used	2 GB	32,767 B	64 bits	100	32768	31
HSQLDB	64 TB	Unlimited[8]	Unlimited[8]	Unlimited[8]	64 TB[7]	Unlimited[8]	Unlimited[8]	0001-01-01	9999-12-31	128
H2	64 TB	Unlimited[8]	Unlimited[8]	Unlimited[8]	64 TB[7]	Unlimited[8]	64 bits	-99999999	99999999	Unlimited[8]
	Max DB size	**Max table size**	**Max row size**	**Max columns per row**	**Max Blob/Clob size**	**Max CHAR size**	**Max NUMBER size**	**Min DATE value**	**Max DATE value**	**Max column name size**

Informix Dynamic Server	~128 PB	~128 PB	32,765 bytes (exclusive of large objects)	32,765	4 TB	32,765	10^{32}	01/01/0001[10]	12/31/9999	128 bytes
Ingres	Unlimited	Unlimited	256 KB	1,024	2 GB	32 000 B	64 bits	0001	9999	256
InterBase	Unlimited[1]	~32 TB	65,536 B	Depends on data types used	2 GB	32,767 B	64 bits	100	32768	31
Linter SQL RDBMS	Unlimited	2^{30} rows	64 KB (w/o BLOBs), 4 GB (BLOB)	250	4 GB	4 KB	64 bits	0001-01-01	9999-12-31	66
Microsoft Access (JET)	2 GB	2 GB	16 MB	255	64 KB (memo field), 1 GB ("OLE Object" field)	255 B (text field)	32 bits	0100	9999	64
Microsoft Visual Foxpro	Unlimited	2 GB	65,500 B	255	2 GB	16 MB	32 bits	0001	9999	10
Microsoft SQL Server	524,272 TB (32 767 files * 16 TB max file size)	524,272 TB	8,060 bytes (Unlimited)[6]	30,000	2 GB	2 GB[6]	126 bits[2]	0001	9999	128
Microsoft SQL Server Compact (Embedded Database)	4 GB	4 GB	8,060 bytes	1024	2 GB	4000	154 bits	0001	9999	128
MySQL 5	Unlimited	MyISAM storage limits: 256 TB; Innodb storage limits: 64 TB	64 KB[3]	4,096[4]	4 GB (longtext, longblob)	64 KB (text)	64 bits	1000	9999	64
OpenLink Virtuoso	32 TB per instance (Unlimited via elastic cluster)	DB size (or 32 TB)	4 KB	200	2 GB	2 GB	2^{31}	0	9999	100
Oracle	Unlimited (4 GB * block size per tablespace)	4 GB * block size (with BIGFILE tablespace)	8 KB	1,000	Unlimited	32,767 B[11]	126 bits	„4712	9999	30
	Max DB size	**Max table size**	**Max row size**	**Max columns per row**	**Max Blob/Clob size**	**Max CHAR size**	**Max NUMBER size**	**Min DATE value**	**Max DATE value**	**Max column name size**
Pervasive PSQL	4 billion objects	256ƒGB	2ƒGB	1,536	2ƒGB	8,000ƒbytes	64ƒbits	01-01-0001	12-31-9999	128ƒbytes

Polyhedra	Limited by available RAM, address space	2^{32} rows	Unlimited	65,536	4ƒGB (subject to RAM)	4ƒGB (subject to RAM)	32ƒbits	0001-01-01	8000-12-31	255
PostgreSQL	Unlimited	32ƒTB	1.6ƒTB	250€1600 depending on type	1ƒGB (text, bytea)[6] - stored inline or 4ƒTB (stored in pg_largeobject)[7]	1ƒGB	Unlimited	„4,713	5,874,897	63
RDM Embedded	Unlimited	2^{48}-1 rows	32ƒKB	1,000	4ƒGB	char: 256, varchar: 4 KB	64ƒbits	0001-01-01	11758978-12-31	31
RDM Server	Unlimited	2^{64}-1 rows	32ƒKB	32,768	Unlimited	32ƒKB	64ƒbits	0001-01-01	11758978-12-31	32
ScimoreDB	Unlimited	16 EB	8,050 B	255	16 TB	8,000 B	64 bits	?	?	?
SQL Anywhere	104 TB (13 files, each file up to 8 TB (32 KB pages))	Limited by file size	Limited by file size	45,000	2 GB	2 GB	64 bits	0001-01-01	9999-12-31	?
SQLite	128 TB (2^{31} pages * 64 KB max page size)	Limited by file size	Limited by file size	32,767	2 GB	2 GB	64 bits	No DATE type[9]	No DATE type[9]	Unlimited
Teradata	Unlimited	Unlimited	64 KB wo/lobs (64 GB w/lobs)	2,048	2 GB	10,000	64 bits	?	9999-12-31 Select 80991231 (date);	30
UniVerse	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited
Xeround Cloud Database	Unlimited	Unlimited	32 GB, depending on available memory	1,000	4 GB	64 KB	64 bits	1000	9999	64
	Max DB size	**Max table size**	**Max row size**	**Max columns per row**	**Max Blob/Clob size**	**Max CHAR size**	**Max NUMBER size**	**Min DATE value**	**Max DATE value**	**Max column name size**

Note (1): Firebird 2.x maximum database size is effectively unlimited with the largest known database size >980 GB. Firebird 1.5.x maximum database size: 32 TB.

Note (2): Limit is 10^{38} using DECIMAL datatype.

Note (3): InnoDB is limited to 8,000 bytes (excluding VARBINARY, VARCHAR, BLOB, or TEXT columns).

Note (4): InnoDB is limited to 1,000 columns.

Note (6): Using VARCHAR (MAX) in SQL 2005 and later.

Note (7): When using a page size of 32 KB, and when BLOB/CLOB data is stored in the database file.

Note (8): Java array size limit of 2,147,483,648 (2^{31}) objects per array applies. This limit applies to number of characters in names, rows per table, columns per table, and characters per CHAR/VARCHAR.

Note (9): Despite the lack of a date datatype, SQLite does include date and time functions, which work for timestamps between 0000-01-01 00:00:00 and 5352-11-01 10:52:47.

Note (10): Informix DATETIME type has adjustable range from YEAR only through 1/10000th second. DATETIME date range is 0001-01-01 00:00:00.00000 through 9999-12-31 23:59:59.99999.

Note (11): Since version 12c. Earlier versions support up to 4000 B.

Tables and views

Information about what tables and views (other than basic ones) are supported natively.

	Temporary table	Materialized view
4th Dimension	Yes	Planned for inclusion in next major release
ADABAS	*?*	*?*
Adaptive Server Enterprise	Yes[1]	Yes - see precomputed result sets
Advantage Database Server	Yes	No (only common views)
Altibase	Yes	No (only common views)
Apache Derby	Yes	No
Clustrix	Yes	No
CUBRID	No	No
Drizzle	Yes	No[4]
DB2	Yes	Yes
Empress Embedded Database	Yes	Yes
EXASolution	Yes	No
Firebird	Yes	No (only common views)
HSQLDB	Yes	No
H2	Yes	No
Informix Dynamic Server	Yes	No[2]
Ingres	Yes	Planned for inclusion in next major release
InterBase	Yes	No
Linter SQL RDBMS	Yes	Yes
LucidDB	No	No
MaxDB	Yes	No
Microsoft Access (JET)	No	No
Microsoft Visual Foxpro	Yes	Yes
Microsoft SQL Server	Yes	Yes[3]
Microsoft SQL Server Compact (Embedded Database)	Yes	No
MonetDB/SQL	Yes	No
MySQL	Yes	No[4]
OpenBase SQL	Yes	Yes
Oracle	Yes	Yes

Oracle Rdb	Yes	Yes
OpenLink Virtuoso	Yes	Yes
Pervasive PSQL	Yes	No
Polyhedra DBMS	No	No (only common views)
PostgreSQL	Yes	Yes[5]
RDM Embedded	Yes	No
RDM Server	Yes	No
SQL Anywhere	Yes	Yes
ScimoreDB	No	No
SQLite	Yes	No
Teradata	Yes	Yes
UniData	Yes	No
UniVerse	Yes	No
Xeround Cloud Database	Yes	No

Note (1): Server provides tempdb, which can be used for public and private (for the session) temp tables.

Note (2): Materialized views are not supported in Informix; the term is used in IBM,s documentation to refer to a temporary table created to run the view,s query when it is too complex, but one cannot for example define the way it is refreshed or build an index on it. The term is defined in the Informix Performance Guide.

Note (3): Query optimizer support only in Developer and Enterprise Editions. In other versions, a direct reference to materialized view and a query hint are required.

Note (4): Materialized views can be emulated using stored procedures and triggers.

Note (5): Materialized views are now standard but can be emulated in versions prior to 9.3 with stored procedures and triggers using PL/pgSQL, PL/Perl, PL/Python, or other procedural languages.

Indexes

Information about what indexes (other than basic B-/B+ tree indexes) are supported natively.

	R-/R+ tree	Hash	Expression	Partial	Reverse	Bitmap	GiST	GIN	Full-text	Spatial	FOT
4th Dimension	*?*	Cluster	*?*	*?*	*?*	*?*	*?*	*?*	Yes	*?*	*?*
ADABAS	*?*	*?*	*?*	*?*	*?*	*?*	*?*	*?*	*?*	*?*	*?*
Adaptive Server Enterprise	No	No	Yes	No	Yes	No	No	No	Yes	*?*	*?*
Advantage Database Server	No	No	Yes	No	Yes	Yes	No	No	Yes	*?*	*?*
Apache Derby	No	No	No	No	No	No	No	No	No	*?*	*?*
Clustrix	No	Yes	No	No	No	No	No	No	No	No	*?*
CUBRID	No	No	Yes	Yes	Yes	No	No	No	*?*	*?*	*?*
Drizzle	No	No	No	No	No	No	No	No	No	*?*	*?*
DB2	No	*?*	Yes	No	Yes	Yes	No	No	Yes	*?*	*?*

Empress Embedded Database	Yes	No	No	Yes	No	Yes	No	No	No	?	?
EXASolution	No	Yes	No	No	No	No	No	No	No	?	?
Firebird	No	No	Yes	No	Yes[1]	No	No	No	No	?	?
HSQLDB	No	No	No	No	No	No	No	No	No	?	?
H2	No	Yes	No	No	No	No	No	No	Yes	?	?
Informix Dynamic Server	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Ingres	Yes	Yes	Ingres v10	No	No	Ingres v10	No	No	No	?	?
InterBase	No	No	No	No	No	No	No	No	No	?	?
Linter SQL RDBMS[10]	No	No	No	No	No	No	No	No	Yes	No	No
LucidDB	No	No	No	No	No	Yes	No	No	No	?	?
MaxDB	No	No	No	No	No	No	No	No	No	?	?
Microsoft Access (JET)	No	No	No	No	No	No	No	No	No	?	?
Microsoft Visual Foxpro	No	No	Yes	Yes	Yes[2]	Yes	No	No	No	?	?
Microsoft SQL Server	?	Non/Cluster & fill factor	Yes[3]	Yes[4]	No[3]	No	No	No	Yes	Yes	?
Microsoft SQL Server Compact (Embedded Database)	No	No	No	No	No	No	No	No	No	?	?
MonetDB/SQL	No	Yes	No	No	No	No	No	No	?	?	?
MySQL	MyISAM tables only	MEMORY, Cluster (NDB), InnoDB,[5] tables only	No	No	No	No	No	No	MyISAM tables and, since v5.6.4, InnoDB tables	MyISAM tables only	?
Oracle	Yes[11]	Cluster Tables	Yes	Yes[6]	Yes	Yes	No	No	Yes	Yes	?
Oracle Rdb	No	Yes	?	No	No	?	No	No	?	?	?
OpenLink Virtuoso	Yes	Cluster	Yes	Yes	No	Yes	No	No	Yes	Yes (Commercial only)	No
Pervasive PSQL	No	No	No	No	No	No	No	No	No	No	No
Polyhedra DBMS	No	Yes	No	No	No	No	No	No	No	No	?
PostgreSQL	Yes	Yes	Yes	Yes	Yes[7]	Yes[8]	Yes	Yes	Yes	PostGIS	?
RDM Embedded	No	Yes	No	Yes	Yes	No	No	No	No	No	No
RDM Server	No	No	No	Yes	Yes	No	No	No	No	No	No
ScimoreDB	No	No	No	No	No	No	No	No	Yes	?	?
SQL Anywhere	No	No	No	No	No	No	No	No	Yes	?	?

SQLite	Yes[8]	No	No	Yes[9]	Yes	No	No	No	Yes[10]	SpatiaLite	?
Teradata	No	Yes	Yes	Yes	No	Yes	No	No	?	?	?
UniVerse	Yes	Yes	Yes[3]	Yes[3]	Yes[3]	No	No	No	?	Yes[11]	?
Xeround Cloud Database	No	Yes	No	No	No	No	No	No	No	No	?
	R-/R+ tree	**Hash**	**Expression**	**Partial**	**Reverse**	**Bitmap**	**GiST**	**GIN**	**Full-text**	**Spatial**	**FOT**

Note (1): The users need to use a function from freeAdhocUDF library or similar.

Note (2): Can be implemented for most data types using expression-based indexes.

Note (3): Can be emulated by indexing a computed column (doesn't easily update) or by using an "Indexed View" (proper name not just any view works).

Note (4): Can be implemented by using an indexed view.

Note (5): InnoDB automatically generates adaptive hash index entries as needed.

Note (6): Can be implemented using Function-based Indexes in Oracle 8i and higher, but the function needs to be used in the sql for the index to be used.

Note (7): A PostgreSQL functional index can be used to reverse the order of a field.

Note (8): PostgreSQL will likely support on-disk bitmap indexes in a future version. Version 8.2 supports a related technique known as "in-memory bitmap scans".

Note (10): B+ tree and full-text only for now.

Note (11): R-Tree indexing available in base edition with Locator but some functionality requires Personal Edition or Enterprise Edition with Spatial option.

Database capabilities

	Union	**Intersect**	**Except**	**Inner joins**	**Outer joins**	**Inner selects**	**Merge joins**	**Blobs and Clobs**	**Common Table Expressions**	**Windowing Functions**	**Parallel Query**
4th Dimension	Yes	Yes	Yes	Yes	Yes	No	No	Yes	?	?	?
ADABAS	Yes	?	?	?	?	?	?	?	?	?	?
Adaptive Server Enterprise	Yes	?	?	Yes	Yes	Yes	Yes	Yes	?	?	Yes
Advantage Database Server	Yes	No	No	Yes	Yes	Yes	Yes	Yes	?	No	?
Altibase	Yes	Yes	Yes, via MINUS	Yes	Yes	Yes	Yes	Yes	No	No	No
Apache Derby	Yes	Yes	Yes	Yes	Yes	?	?	Yes	No	No	?
Clustrix	Yes	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
CUBRID	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	?
Drizzle	Yes	No	No	Yes	Yes	Yes	No	Yes	No	No	No[12]
DB2	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Empress Embedded Database	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	?	?	?
EXASolution	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
Firebird	Yes	?	?	Yes	Yes	Yes	Yes	Yes	Yes	Yes	?
HSQLDB	Yes	Yes	Yes	Yes	Yes	Yes	Yes[13]	Yes	Yes	No	Yes
H2	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	experimental[14]	No[15]	?
Informix Dynamic Server	Yes	Yes	Yes, via MINUS	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes[16]
Ingres	Yes	No	No	Yes	Yes	Yes	Yes	Yes	No	No	?
InterBase	Yes	?	?	Yes	Yes	?	?	Yes	?	?	?
Linter SQL RDBMS	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
LucidDB	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	?	?	?
MaxDB	Yes	?	?	Yes	Yes	Yes	No	Yes	?	?	?
Microsoft Access (JET)	Yes	No	No	Yes	Yes	Yes	No	Yes	No	No	?
Microsoft Visual Foxpro	Yes	?	?	Yes	Yes	Yes	?	Yes	?	?	?
Microsoft SQL Server	Yes	Yes (2005 and beyond)	Yes (2005 and beyond)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Microsoft SQL Server Compact (Embedded Database)	Yes	No	No	Yes	Yes	?	No	Yes	No	No	?
MonetDB/SQL	?	?	?	?	?	?	?	?	?	?	?
MySQL	Yes	No	No	Yes	Yes	Yes	No	Yes	No	No	No
OpenBase SQL	No	No	No	Yes	Yes	Yes	Yes	Yes	?	?	?
OpenLink Virtuoso	Yes	?	?	Yes	Yes	Yes	?	Yes	?	?	?
Oracle	Yes	Yes	Yes, via MINUS	Yes	Yes	Yes	Yes	Yes	Yes [1]	Yes	Yes
Oracle Rdb	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	?	?	?
Pervasive PSQL	Yes	No	No	Yes	Yes	?	?	Yes	No	No	No
Polyhedra DBMS	Yes	Yes	Yes	Yes	No	?	?	Yes	?	?	?
PostgreSQL	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
RDM Embedded	No	No	No	Yes	Yes	No	No	Yes	No	No	No
RDM Server	Yes	No	No	Yes	Yes	Yes	No	Yes	No	No	No
ScimoreDB	Yes	?	?	Yes	LEFT only	Yes	Yes	Yes	?	?	?
SmallSQL	?	?	?	?	?	?	?	?	?	?	?
SQL Anywhere	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

SQLite	Yes	Yes	Yes	Yes	LEFT only	Yes	No	Yes	No	No	No
Teradata	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
UniVerse	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	?
Xeround Cloud Database	Yes	No	No	Yes	Yes	Yes	No	Yes	No	No	No
	Union	**Intersect**	**Except**	**Inner joins**	**Outer joins**	**Inner selects**	**Merge joins**	**Blobs and Clobs**	**Common Table Expressions**	**Windowing Functions**	**Parallel Query**

Note (1): Recursive CTEs introduced in 11gR2 supersedes similar construct called CONNECT BY.

Data types

	Type system	Integer	Floating point	Decimal	String	Binary	Date/Time	Boolean	Other
4th Dimension	Static	UUID (16-bit), SMALLINT (16-bit), INT (32-bit), BIGINT (64-bit), NUMERIC (64-bit)	REAL, FLOAT	REAL, FLOAT	CLOB, TEXT, VARCHAR	BIT, BIT VARYING, BLOB	DURATION, INTERVAL, TIMESTAMP	BOOLEAN	PICTURE
Altibase	Static	SMALLINT (16-bit), INTEGER (32-bit), BIGINT (64-bit)	REAL(32-bit), DOUBLE(64-bit)	DECIMAL, NUMERIC, NUMBER, FLOAT	CHAR, VARCHAR, NCHAR, NVARCHAR, CLOB	BLOB, BYTE, NIBBLE, BIT, VARBIT	DATE		GEOMETRY
Clustrix	Static	TINYINT (8-bit), SMALLINT (16-bit), MEDIUMINT (24-bit), INT (32-bit), BIGINT (64-bit)	FLOAT (32-bit), DOUBLE	DECIMAL	CHAR, BINARY, VARCHAR, VARBINARY, TEXT, TINYTEXT, MEDIUMTEXT, LONGTEXT	TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB	DATETIME, DATE, TIMESTAMP, YEAR	BIT(1), BOOLEAN	ENUM, SET,
CUBRID	Static	SMALLINT (16-bit), INTEGER (32-bit), BIGINT (64-bit)	FLOAT, REAL(32-bit), DOUBLE(64-bit)	DECIMAL, NUMERIC	CHAR, VARCHAR, NCHAR, NVARCHAR, CLOB	BLOB	DATE, DATETIME, TIME, TIMESTAMP	BIT	MONETARY, BIT VARYING, SET, MULTISET, SEQUENCE, ENUM
Drizzle	Static	INT (32-bit), BIGINT (64-bit)	DOUBLE (aka REAL) (64-bit)	DECIMAL	BINARY, VARCHAR, VARBINARY, TEXT,	BLOB	DATETIME, DATE, TIMESTAMP		ENUM, SERIAL

Empress Embedded Database	Static	TINYINT, SQL_TINYINT, or INTEGER8; SMALLINT, SQL_SMALLINT, or INTEGER16; INTEGER, INT, SQL_INTEGER, or INTEGER32; BIGINT, SQL_BIGINT, or INTEGER64	REAL, SQL_REAL, or FLOAT32; DOUBLE PRECISION, SQL_DOUBLE, or FLOAT64; FLOAT, or SQL_FLOAT; EFLOAT	DECIMAL, DEC, NUMERIC, SQL_DECIMAL, or SQL_NUMERIC; DOLLAR	CHARACTER, ECHARACTER, CHARACTER VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, NLSCHARACTER, CHARACTER LARGE OBJECT, TEXT, NATIONAL CHARACTER LARGE OBJECT, NLSTEXT	BINARY LARGE OBJECT or BLOB; BULK	DATE, EDATE, TIME, ETIME, EPOCH_TIME, TIMESTAMP, MICROTIMESTAMP	BOOLEAN	SEQUENCE 32, SEQUENCE
EXASolution	Static	TINYINT, SMALLINT, INTEGER, BIGINT,	REAL, FLOAT, DOUBLE	DECIMAL, DEC, NUMERIC, NUMBER	CHAR, NCHAR, VARCHAR, VARCHAR2, NVARCHAR, NVARCHAR2, CLOB, NCLOB	N/A	DATE, TIMESTAMP, INTERVAL	BOOLEAN, BOOL	GEOMETRY
HSQLDB	Static	TINYINT (8-bit), SMALLINT (16-bit), INTEGER (32-bit), BIGINT (64-bit)	DOUBLE (64-bit)	DECIMAL, NUMERIC	CHAR, VARCHAR, LONGVARCHAR, CLOB	BINARY, VARBINARY, LONGVARBINARY, BLOB	DATE, TIME, TIMESTAMP, INTERVAL	BOOLEAN	OTHER (object), BIT, BIT VARYING, ARRAY
Informix Dynamic Server	Static	SMALLINT (16-bit), INT (32-bit), INT8 (64-bit proprietary), BIGINT (64-bit)	SMALLFLOAT (32-bit), FLOAT (64-bit)	DECIMAL (32 digits float/fixed), MONEY	CHAR, VARCHAR, NCHAR, NVARCHAR, LVARCHAR, CLOB, TEXT	TEXT, BYTE, BLOB, CLOB	DATE, DATETIME, INTERVAL	BOOLEAN	SET, LIST, MULTISET, ROW, TIMESERIES, SPATIAL, USER DEFINED TYPES
Ingres	Static	TINYINT (8-bit), SMALLINT (16-bit), INTEGER (32-bit), BIGINT (64-bit)	FLOAT4 (32-bit), FLOAT (64-bit)	DECIMAL	C, CHAR, VARCHAR, LONG VARCHAR, NCHAR, NVARCHAR, LONG NVARCHAR, TEXT	BYTE, VARBYTE, LONG VARBYTE (BLOB)	DATE, ANSIDATE, INGRESDATE, TIME, TIMESTAMP, INTERVAL	N/A	MONEY, OBJECT_KEY, TABLE_KEY, USER-DEFINED DATA TYPES (via OME)
Linter SQL RDBMS	Static	SMALLINT (16-bit), INTEGER (32-bit), BIGINT (64-bit)	REAL(32-bit), DOUBLE(64-bit)	DECIMAL, NUMERIC	CHAR, VARCHAR, NCHAR, NVARCHAR, BLOB	BYTE, VARBYTE, BLOB	DATE	BOOLEAN	GEOMETRY, EXTFILE
Microsoft SQL Server	Static	TINYINT, SMALLINT, INT, BIGINT	FLOAT, REAL	NUMERIC, DECIMAL, SMALLMONEY, MONEY	CHAR, VARCHAR, TEXT, NCHAR, NVARCHAR, NTEXT	BINARY, VARBINARY, IMAGE, FILESTREAM	DATE, DATETIMEOFFSET, DATETIME2, SMALLDATETIME, DATETIME, TIME	BIT	CURSOR, TIMESTAMP, HIERARCHYID, UNIQUEIDENTIFIER, SQL_VARIANT, XML, TABLE
Microsoft SQL Server Compact (Embedded Database)	Static	TINYINT, SMALLINT, INT, BIGINT	FLOAT, REAL	NUMERIC, DECIMAL, MONEY	NCHAR, NVARCHAR, NTEXT	BINARY, VARBINARY, IMAGE	DATETIME	BIT	TIMESTAMP, ROWVERSION, UNIQUEIDENTIFIER, IDENTITY, ROWGUIDCOL

MySQL	Static	TINYINT (8-bit), SMALLINT (16-bit), MEDIUMINT (24-bit), INT (32-bit), BIGINT (64-bit)	FLOAT (32-bit), DOUBLE (aka REAL) (64-bit)	DECIMAL	CHAR, BINARY, VARCHAR, VARBINARY, TEXT, TINYTEXT, MEDIUMTEXT, LONGTEXT	TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB	DATETIME, DATE, TIMESTAMP, YEAR	BIT(1), BOOLEAN (aka BOOL) = synonym for TINYINT	ENUM, SET, GIS data types (Geometry, Point, Curve, LineString, Surface, Polygon, GeometryCollection, MultiPoint, MultiCurve, MultiLineString, MultiSurface, MultiPolygon)
OpenLink Virtuoso	Static + Dynamic	INT, INTEGER, SMALLINT	REAL, DOUBLE PRECISION, FLOAT, FLOAT('INTNUM')'	DECIMAL, DECIMAL'('INTNUM')', DECIMAL'('INTNUM','INTNUM')', NUMERIC, NUMERIC'('INTNUM')', NUMERIC'('INTNUM','INTNUM')'	CHARACTER, CHAR'('INTNUM')', VARCHAR, VARCHAR'('INTNUM')', NVARCHAR, NVARCHAR'('INTNUM')'	BLOB	TIMESTAMP, DATETIME, TIME, DATE	n/a	GEOMETRY, REFERENCE (URI), UDT (User Defined Type)
Oracle	Static + Dynamic (through ANYDATA)	NUMBER	BINARY_FLOAT, BINARY_DOUBLE	NUMBER	CHAR, VARCHAR2, CLOB, NCLOB, NVARCHAR2, NCHAR, LONG (deprecated)	BLOB, RAW, LONG RAW (deprecated), BFILE	DATE, TIMESTAMP (with/without TIMEZONE), INTERVAL	N/A	SPATIAL, IMAGE, AUDIO, VIDEO, DICOM, XMLType
Pervasive PSQL	Static	BIGINT, INTEGER, SMALLINT, TINYINT, UBIGINT, UINTEGER, USMALLINT, UTINYINT	BFLOAT4, BFLOAT8, DOUBLE, FLOAT	DECIMAL, NUMERIC, NUMERICSA, NUMERICSLB, NUMERICSLS, NUMERICSTB, NUMERICSTS	CHAR, LONGVARCHAR, VARCHAR	BINARY, LONGVARBINARY, VARBINARY	DATE, DATETIME, TIME	BIT	CURRENCY, IDENTITY, SMALLIDENTITY, TIMESTAMP, UNIQUEIDENTIFIER
Polyhedra	Static	INTEGER8 (8-bit), INTEGER(16-bit), INTEGER (32-bit), INTEGER64 (64-bit)	FLOAT32 (32-bit), FLOAT (aka REAL; 64-bit)	N/A	VARCHAR, LARGE VARCHAR (aka CHARACTER LARGE OBJECT)	LARGE BINARY (aka BINARY LARGE OBJECT)	DATETIME	BOOLEAN	N/A
PostgreSQL	Static	SMALLINT (16-bit), INTEGER (32-bit), BIGINT (64-bit)	REAL (32-bit), DOUBLE PRECISION (64-bit)	DECIMAL, NUMERIC	CHAR, VARCHAR, TEXT	BYTEA	DATE, TIME (with/without TIMEZONE), TIMESTAMP (with/without TIMEZONE), INTERVAL	BOOLEAN	ENUM, POINT, LINE, LSEG, BOX, PATH, POLYGON, CIRCLE, CIDR, INET, MACADDR, BIT, UUID, XML, JSON, arrays, composites, ranges, custom
RDM Embedded	Static	tinyint, smallint, integer, bigint	real, float, double	N/A	char, varchar, wchar, varwchar, long varchar, long varwchar	binary, varbinary, long varbinary	date, time, timestamp	bit	N/A
RDM Server	Static	tinyint, smallint, integer, bigint	real, float, double	decimal, numeric	char, varchar, wchar, varwchar, long varchar, long varwchar	binary, varbinary, long varbinary	date, time, timestamp	bit	rowid
SQLite	Dynamic	INTEGER (64-bit)	REAL (aka FLOAT, DOUBLE) (64-bit)	N/A	TEXT (aka CHAR, CLOB)	BLOB	N/A	N/A	N/A
UniData	Dynamic	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
UniVerse	Dynamic	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Xeround Cloud Database	Static	TINYINT (8-bit), SMALLINT (16-bit), MEDIUMINT (24-bit), INT (32-bit), BIGINT (64-bit)	FLOAT (32-bit), DOUBLE (aka REAL) (64-bit)	DECIMAL	CHAR, BINARY, VARCHAR, VARBINARY, TEXT, TINYTEXT, MEDIUMTEXT, LONGTEXT	TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB	DATETIME, DATE, TIMESTAMP, YEAR	BOOLEAN (aka BOOL) = synonym for TINYINT	ENUM, SET
	Type system	**Integer**	**Floating point**	**Decimal**	**String**	**Binary**	**Date/Time**	**Boolean**	**Other**

Other objects

Information about what other objects are supported natively.

	Data Domain	Cursor	Trigger	Function[1]	Procedure[1]	External routine[1]
4th Dimension	Yes	No	Yes	Yes	Yes	Yes
ADABAS	?	Yes	?	Yes?	Yes?	Yes
Adaptive Server Enterprise	Yes	Yes	Yes	Yes	Yes	Yes
Advantage Database Server	Yes	Yes	Yes	Yes	Yes	Yes
Altibase	Yes	Yes	Yes	Yes	Yes	Yes
Apache Derby	No	Yes	Yes	Yes[2]	Yes[2]	Yes[2]
Clustrix	No	Yes	No	Yes	Yes	Yes
CUBRID	Yes	Yes	Yes	Yes	Yes[2]	Yes
Drizzle	Yes	Yes	Yes[4]	Yes[4]	Yes[4]	Yes[4]
Empress Embedded Database	Yes via RANGE CHECK	Yes	Yes	Yes	Yes	Yes
EXASolution	Yes	No	No	Yes	Yes	Yes
DB2	Yes via CHECK CONSTRAINT	Yes	Yes	Yes	Yes	Yes
Firebird	Yes	Yes	Yes	Yes	Yes	Yes
HSQLDB	Yes	No	Yes	Yes	Yes	Yes
H2	Yes	No	Yes[2]	Yes[2]	Yes[2]	Yes
Informix Dynamic Server	Yes via CHECK	Yes	Yes	Yes	Yes	Yes [5]
Ingres	Yes	Yes	Yes	Yes	Yes	Yes
InterBase	Yes	Yes	Yes	Yes	Yes	Yes
Linter SQL RDBMS	No	Yes	Yes	Yes	Yes	No
LucidDB	No	Yes	No	Yes[2]	Yes[2]	Yes[2]
MaxDB	Yes	Yes	Yes	Yes	Yes	?
Microsoft Access (JET)	Yes	No	No	No	Yes, But single DML/DDL Operation	Yes
Microsoft Visual Foxpro	No	Yes	Yes	Yes	Yes	Yes
Microsoft SQL Server	Yes (2000 and beyond)	Yes	Yes	Yes	Yes	Yes

Microsoft SQL Server Compact (Embedded Database)	No	Yes	No	No	No	No
MonetDB	No	No	Yes	Yes	Yes	Yes
MySQL	No [3]	Yes	Yes	Yes	Yes	Yes
OpenBase SQL	Yes	Yes	Yes	Yes	Yes	Yes
Oracle	Yes	Yes	Yes	Yes	Yes	Yes
Oracle Rdb	Yes	Yes	Yes	Yes	Yes	Yes
OpenLink Virtuoso	Yes	Yes	Yes	Yes	Yes	Yes
Pervasive PSQL	Yes	Yes	Yes	Yes	Yes	No
Polyhedra DBMS	No	No	Yes	Yes	Yes	Yes
PostgreSQL	Yes	Yes	Yes	Yes	Yes	Yes
RDM Embedded	No	Yes	No	No	Yes	Yes
RDM Server	No	Yes	Yes	No	Yes	Yes
ScimoreDB	No	No	No	No	Yes	Yes
SQL Anywhere	Yes	Yes	Yes	Yes	Yes	Yes
SQLite	No	No	Yes	No	No	Yes
Teradata	No	Yes	Yes	Yes	Yes	Yes
UniData	No	No	Yes	Yes	Yes	Yes
UniVerse	No	No	Yes	Yes	Yes	Yes
Xeround Cloud Database	No [3]	Yes	Yes	Yes	Yes	No
	Data Domain	**Cursor**	**Trigger**	**Function** [1]	**Procedure** [1]	**External routine** [1]

Note (1): Both **function** and **procedure** refer to internal routines written in SQL and/or procedural language like PL/SQL. **External routine** refers to the one written in the host languages, such as C, Java, Cobol, etc. "Stored procedure" is a commonly used term for these routine types. However, its definition varies between different database vendors.

Note (2): In Derby, H2, LucidDB, and CUBRID, users code **functions** and **procedures** in Java.

Note (3): ENUM datatype exist. CHECK clause is parsed, but not enforced in runtime.

Note (4): In Drizzle the user codes **functions** and **procedures** in C++.

Note (5): Informix supports external functions written in Java, C, & C++.

Partitioning

Information about what partitioning methods are supported natively.

	Range	Hash	Composite (Range+Hash)	List	Expression
4th Dimension	?	?	?	?	?
ADABAS	?	?	?	?	?
Adaptive Server Enterprise	Yes	Yes	No	Yes	?
Advantage Database Server	No	No	No	No	?
Altibase	Yes	Yes	No	Yes	?
Apache Derby	No	No	No	No	?
Clustrix	Yes	No	No	No	No
CUBRID	Yes	Yes	No	Yes	?
IBM DB2	Yes	Yes	Yes	Yes	?
Empress Embedded Database	No	No	No	No	?
EXASolution	No	Yes	No	No	No
Firebird	No	No	No	No	?
HSQLDB	No	No	No	No	?
H2	No	No	No	No	?
Informix Dynamic Server	Yes	Yes	Yes	Yes	Yes
Ingres	Yes	Yes	Yes	Yes	?
InterBase	No	No	No	No	?
Linter SQL RDBMS	No	No	No	No	?
MaxDB	No	No	No	No	?
Microsoft Access (JET)	No	No	No	No	?
Microsoft Visual Foxpro	No	No	No	No	?
Microsoft SQL Server	Yes	No	No	No	?
Microsoft SQL Server Compact (Embedded Database)	No	No	No	No	?
MonetDB	Yes (M5)	Yes (M5)	Yes (M5)	No	?
MySQL	Yes	Yes	Yes	Yes	?
OpenBase SQL	?	?	?	?	?
Oracle	Yes	Yes	Yes	Yes	?
Oracle Rdb	Yes	Yes	?	?	?
OpenLink Virtuoso	Yes	Yes	Yes	Yes	Yes
Pervasive PSQL	No	No	No	No	No
Polyhedra DBMS	No	No	No	No	No
PostgreSQL	Yes[1]	Yes[1]	Yes[1]	Yes[1]	?
RDM Embedded	Yes[2]	Yes[2]	Yes[2]	No	?

RDM Server	No	No	No	No	**?**
ScimoreDB	No	Yes	No	No	**?**
SQL Anywhere	No	No	No	No	**?**
SQLite	No	No	No	No	**?**
Teradata	Yes	Yes	Yes	Yes	**?**
UniVerse	Yes	Yes	Yes	Yes	**?**
Xeround Cloud Database	N/A - partitioning provided transparently	N/A - partitioning provided transparently	N/A - partitioning provided transparently	N/A - partitioning provided transparently	N/A - partitioning provided transparently
	Range	**Hash**	**Composite (Range+Hash)**	**List**	**Expression**

Note (1): PostgreSQL 8.1 provides partitioning support through check constraints. Range, List and Hash methods can be emulated with PL/pgSQL or other procedural languages.

Note (2): RDM Embedded 10.1 requires the application programs to select the correct partition (using range, hash or composite techniques) when adding data, but the *database union* functionality allows all partitions to be read as a single database.

Access control

Information about access control functionalities (*work in progress*).

	Native network encryption[1]	**Brute-force protection**	**Enterprise directory compatibility**	**Password complexity rules**[2]	**Patch access**[3]	**Run unprivileged**[4]	**Audit**	**Resource limit**	**Separation of duties (RBAC)**[5]	**Security Certification**	**Label Based Access Control (LBAC)**
4D	Yes (with SSL)	**?**	**?**	**?**	**?**	**?**	**?**	**?**	**?**	**?**	**?**
Adaptive Server Enterprise	Yes (optional; to pay)	Yes	Yes (optional ?)	Yes	Partial (need to register; depend on which product)	Yes	Yes	Yes	Yes	Yes (EAL4+/[1])	**?**
Advantage Database Server	Yes	No	No	No	Yes	Yes	No	No	Yes	**?**	**?**
DB2	Yes	**?**	Yes (LDAP, Kerberos ‘)	Yes	**?**	Yes	Yes	Yes	Yes	Yes (EAL4+[6])	**?**
Empress Embedded Database	**?**	**?**	No	No	Yes	Yes	Yes	No	Yes	No	**?**
EXASolution	No	No	Yes (LDAP)	No	Yes	Yes	Yes	Yes	Yes	No	**?**
Firebird	No	Yes	Yes (Windows trusted authenification)	No	Partial (no security page)	Yes	No	No	No[7]	**?**	**?**
HSQLDB	Yes	No	Yes	Yes	Yes	Yes	No	No	Yes	No	**?**

H2	Yes	Yes	?	No	?	Yes	?	Yes	Yes	No	?
Informix Dynamic Server	Yes	?	Yes[10]	?[10]	Yes	Yes	Yes	Yes	Yes	?	Yes
Linter SQL RDBMS	Yes (with SSL)	Yes	Yes	Yes (length only)	No	Yes	Yes	Yes	Yes	Yes	Yes
MariaDB	Yes (SSL)	No	Yes (with 5.2, but not on Windows servers)	No	Partial (no security page)	Yes	?	?	?[8]	No	?
Microsoft SQL Server	Yes	?	Yes (Microsoft Active Directory)	Yes	Yes	Yes	Yes (From 2008)	Yes	Yes	Yes (EAL4+[11])	?
Microsoft SQL Server Compact (Embedded Database)	No (not relevant, only file permissions)	No (not relevant)	No (not relevant)	No (not relevant)	Yes	Yes (file access)	Yes	Yes	No	?	?
MySQL	Yes (SSL with 4.0)	No	Yes (with 5.5, but only in commercial edition)	No	Partial (no security page)	Yes	?	?	?[8]	No	?
OpenBase SQL	Yes	?	Yes (Open Directory, LDAP)	No	?	?	?	?	?	?	?
OpenLink Virtuoso	Yes	Yes	Yes	Yes (optional)	Yes (optional)	Yes	Yes (optional)	Yes (optional)	Yes	No	?
Oracle	Yes	Yes	Yes	Yes	?	Yes	Yes	Yes	Yes	Yes (EAL4+[1])	?
Pervasive PSQL	Yes	?	No	No	Yes	Yes	Yes [12]	No	No	No	?
Polyhedra DBMS	No	No	No	No	No	Yes	Yes [13]	Yes	Yes [13]	No	?
PostgreSQL	Yes	Yes (for 9.1)	Yes (LDAP, Kerberos ‘ [9])	Yes (as of 9.0 with passwordcheck module)	Yes	Yes	No	Yes	Yes	Yes (EAL1[1])	?
RDM Embedded	No	No	No	No	No	Yes	No	No	No	No	?
RDM Server	Yes	No	No	No	No	Yes	Yes	No	Yes	No	?
SQL Anywhere	Yes	?	Yes (Kerberos)	Yes	?	Yes	Yes	No	Yes	Yes (EAL3+[1] as Adaptive Server Anywhere)	?
SQLite	No (not relevant, only file permissions)	No (not relevant)	No (not relevant)	No (not relevant)	Partial (no security page)	Yes (file access)	Yes	Yes	No	No	?

Xeround Cloud Database	Yes (SSL with 4.0)	No	No	No	N/A - database as a service	Yes	No	No	No	No	**?**
	Native network encryption[1]	**Brute-force protection**	**Enterprise directory compatibility**	**Password complexity rules[2]**	**Patch access[3]**	**Run unprivileged[4]**	**Audit**	**Resource limit**	**Separation of duties (RBAC)[5]**	**Security Certification**	**Label Based Access Control (LBAC)**

Note (1): Network traffic could be transmitted in a secure way (not clear-text, in general SSL encryption). Precise if option is default, included option or an extra modules to buy.

Note (2): Options are present to set a minimum size for password, respect complexity like presence of numbers or special characters.

Note (3): How do you get security updates? Is it free access, do you need a login or to pay? Is there easy access through a Web/FTP portal or RSS feed or only through offline access (mail CD-ROM, phone).

Note (4): Does database process run as root/administrator or unprivileged user? What is default configuration?

Note (5): Is there a separate user to manage special operation like backup (only dump/restore permissions), security officer (audit), administrator (add user/create database), etc.? Is it default or optional?

Note (6): Common Criteria certified product list.

Note (7): FirebirdSQL seems to only have SYSDBA user and DB owner. There are no separate roles for backup operator and security administrator.

Note (8): User can define a dedicated backup user but nothing particular in default install.

Note (9): Authentication methods.

Note (10): Informix Dynamic Server supports PAM and other configurable authentication. By default uses OS authentication.

Note (11): Authentication methods.

Note (12): With the use of Pervasive AuditMaster.

Note (13): User-based security is optional in Polyhedra, but when enabled can be enhanced to a role-based model with auditing.

Databases vs schemas (terminology)

The SQL specification makes clear what an "SQL schema" is; however, different databases implement it incorrectly. To compound this confusion the functionality can, when incorrectly implemented, overlap with that of the parent-database. An SQL schema is simply a namespace within a database, things within this namespace are addressed using the member operator dot " . ". This seems to be a universal amongst all of the implementations.

A true fully (database, schema, and table) qualified query is exemplified as such: `SELECT * FROM database.schema.table`

Now, the issue, both a schema and a database can be used to isolate one table, "foo" from another like named table "foo". The following is pseudo code:

- `SELECT * FROM db1.foo` vs. `SELECT * FROM db2.foo` (no explicit schema between db and table)
- `SELECT * FROM [db1.]default.foo` vs. `SELECT * FROM [db1.]alternate.foo` (no explicit db prefix)

The problem that arises is that former MySQL users will create multiple databases for one project. In this context, MySQL databases are analogous in function to Postgres-schemas, insomuch as Postgres lacks off-the-shelf

cross-database functionality that MySQL has. Conversely, PostgreSQL has applied more of the specification implementing cross-table, cross-schema, and then left room for future cross-database functionality.

MySQL aliases *schema* with *database* behind the scenes, such that CREATE SCHEMA and CREATE DATABASE are analogs. It can therefore be said that MySQL has implemented cross-database functionality, skipped schema functionality entirely, and provided similar functionality into their implementation of a database. In summary, Postgres fully supports schemas but lacks some functionality MySQL has with databases, while MySQL does not even attempt to support true schemas.

Oracle has its own spin where creating a user is synonymous with creating a schema. Thus a database administrator can create a user called PROJECT and then create a table PROJECT.TABLE. Users can exist without schema objects, but an object is always associated with an owner (though that owner may not have privileges to connect to the database). With the Oracle 'shared-everything' RAC architecture, the same database can be opened by multiple servers concurrently. This is independent of replication, which can also be used, whereby the data is copied for use by different server. In the Oracle view, the 'database' is a set of files which contains the data while the 'instance' is a set of processes (and memory) through which a database is accessed.

Informix supports multiple databases in a server instance, like MySQL. It supports the CREATE SCHEMA syntax as a way to group DDL statements into a single unit creating all objects created as a part of the schema as a single owner. Informix supports a database mode called ANSI mode which supports creating objects with the same name but owned by different users.

The end result is confusion between the database factions. The Postgres and Oracle communities maintain that one database is all that is needed for one project, per the definition of database. MySQL and Informix proponents maintain that schemas have no legitimate purpose when the functionality can be achieved with databases. Postgres adheres to the SQL specification, in a more intuitive fashion (bottom-up), while MySQL,s pragmatic counterargument allows their users to get the job done while creating conceptual confusion.

References

[1] hsqldb (http://sourceforge.net/projects/hsqldb/files/hsqldb/hsqldb_2_2/)
[2] http://techotv.com/run-apache-mysql-php-http-web-server-android-os-phone-tablet/Run Apache, Mysql, Php € Web server on Android mobile or Tablet
[3] http://www.oss4zos.org/mediawiki/index.php?title=PostgreSQL#z.2FOS
[4] Transactional DDL in PostgreSQL: A Competitive Analysis (http://wiki.postgresql.org/wiki/Transactional_DDL_in_PostgreSQL:_A_Competitive_Analysis)
[5] SQLite Full Unicode support is optional and not installed by default in most systems (http://www.sqlite.org/faq.html#q18) (like Android, Debian ‘)
[6] http://grokbase.com/t/postgresql/pgsql-general/12bsww982c/large-insert-leads-to-invalid-memory-alloc
[7] http://www.postgresql.org/docs/9.3/static/lo-intro.html
[8] The SQLite R*Tree Module (http://www.sqlite.org/rtree.html)
[9] SQLite Partial Indexes (http://sqlite.org/partialindex.html)
[10] SQLite FTS3 Extension (http://www.sqlite.org/fts3.html)
[11] geospatial
[12] How does Drizzle handle parallel "things"? (https://answers.launchpad.net/drizzle/+question/135548)
[13] New Features in HyperSQL 2.2 (http://hsqldb.org/web/features200.html)
[14] H2 > Advanced > Recursive Queries (http://h2database.com/html/advanced.html#recursive_queries)
[15] H2 Roadmap (http://www.h2database.com/html/roadmap.html)
[16] Informix parallel data query (PDQ) (http://portal.acm.org/citation.cfm?id=382443)

External links

- Comparison of different SQL implementations against SQL standards (http://troels.arvin.dk/db/rdbms/). Includes Oracle, DB2, Microsoft SQL Server, MySQL and PostgreSQL. (08/Jun/2007)
- Features, strengths and weaknesses comparison between Oracle and MSSQL (independent). (http://www.wisdomforce.com/resources/docs/MSSQL2005_ORACLE10g_compare.pdf)
- The SQL92 standard (http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt)
- MetaMarket Druid IMDB (http://metamarkets.com/druid/)
- VM-Ware Redis IMDB (http://redis.io/)
- CSQL DB (http://www.csqldb.com)

Document-oriented database

A **document-oriented database** is a computer program designed for storing, retrieving, and managing document-oriented information, also known as semi-structured data. Document-oriented databases are one of the main categories of so-called NoSQL databases and the popularity of the term "document-oriented database" (or "document store") has grown[*citation needed*] with the use of the term NoSQL itself. In contrast to well-known relational databases and their notions of "Relations" (or "Tables"), these systems are designed around an abstract notion of a "Document".

Documents

The central concept of a document-oriented database is the notion of a *Document*. While each document-oriented database implementation differs on the details of this definition, in general, they all assume documents encapsulate and encode data (or information) in some standard formats or encodings. Encodings in use include XML, YAML, JSON, and BSON, as well as binary forms like PDF and Microsoft Office documents (MS Word, Excel, and so on).

Documents inside a document-oriented database are similar, in some ways, to records or rows in relational databases, but they are less rigid. They are not required to adhere to a standard schema, nor will they have all the same sections, slots, parts, or keys. For example, the following is a document:

```
{
   FirstName: "Bob",
   Address: "5 Oak St.",
   Hobby: "sailing"
}
```

A second document might be:

```
{
    FirstName: "Jonathan",
    Address: "15 Wanamassa Point Road",
    Children: [
        {Name: "Michael", Age: 10},
        {Name: "Jennifer", Age: 8},
        {Name: "Samantha", Age: 5},
        {Name: "Elena", Age: 2}
  ]
 }
```

These two documents share some structural elements with one another, but each also has unique elements. Unlike a relational database where every record contains the same fields, leaving unused fields empty; there are no empty 'fields' in either document (record) in the above example. This approach allows new information to be added to some records without requiring that every other record in the database shares the same structure.

Keys

Documents are addressed in the database via a unique *key* that represents that document. This key is often a simple string, a URI, or a path. The key can be used to retrieve the document from the database. Typically, the database retains an index on the key to speed up document retrieval.

Retrieval

Another defining characteristic of a document-oriented database is that, beyond the simple key-document (or key-value) lookup that can be used to retrieve a document, the database offers an API or query language that allows the user to retrieve documents based on their content. For example, you may want a query that retrieves all the documents with a certain field set to a certain value. The set of query APIs or query language features available, as well as the expected performance of the queries, varies significantly from one implementation to the next.

Organization

Implementations offer a variety of ways of organizing documents, including notions of

- Collections
- Tags
- Non-visible Metadata
- Directory hierarchies
- Buckets

Implementations

Name	Publisher	License	Language	Notes	RESTful API
ArangoDB [1]	triAGENS [2]	Apache 2 License	C, C++ & Javascript	A multi model, high-performance document store and graph database.	Yes [3]
BaseX	BaseX Team [4]	BSD License	Java, XQuery	Support for XML, JSON and binary formats; client-/server based architecture; concurrent structural and full-text searches and updates; REST APIs.	Yes
Cassandra	Apache Software Foundation	Apache License	Java	JSON over HTTP	Yes
Cloudant	Cloudant, Inc. [5]	Proprietary	Erlang, Java, Scala, and C	Distributed database service based on BigCouch, the company's open source fork of the Apache-backed CouchDB project.	Yes
Clusterpoint	Clusterpoint Ltd. [6]	Free community license / Commercial [7]	C++	Schema-free, document-oriented database management system platform with server based data storage, full text search engine functionality, information ranking for search relevance and clustering.	Yes
Couchbase Server	Couchbase, Inc.	Apache License	Erlang and C	Distributed NoSQL Document Database.	Yes [8]

CouchDB	Apache Software Foundation	Apache License	Erlang	JSON over REST/HTTP with Multi-Version Concurrency Control and limited ACID properties. Uses map and reduce for views and queries.[9]	Yes [10]
eXist	eXist, [11]	GPL	XQuery, Java	XML over REST/HTTP, WebDAV, Lucene Fulltext search, validation, versioning, clustering, triggers, URL rewriting, collections, ACLS, XQuery Update	Yes [12]
FleetDB	FleetDB [13]	MIT License	Clojure	A JSON-based [14] schema-free database optimized for agile development.	(unknown)
Jackrabbit	Apache Software Foundation	Apache License	Java		(unknown)
Inquire	Infodata Systems, Inc.	Proprietary	unknown	In the mid-80's this was the dominant document-oriented commercial database, widely successful. The company seems to have gone out of business in 2005.	(unknown)
Lotus Notes	IBM	Proprietary	LotusScript, Java, Lotus @Formula		(unknown)
MarkLogic	MarkLogic Corporation	Free Express license [15] or Commercial [15]	REST, Java, XQuery, XSLT, C++	Distributed document-oriented database with Multi-Version Concurrency Control, integrated Full text search and ACID-compliant transaction semantics	Yes
MongoDB	MongoDB, Inc	GNU AGPL v3.0[16]	C++	Document-oriented database optimized for highly transient data	Optional [17]
MUMPS Database[18]		Proprietary and Affero GPL[19]	MUMPS	Commonly used in health applications.	(unknown)
OrientDB	Orient Technologies [20]	Apache License	Java	JSON over HTTP	Yes
RavenDB	Hibernating Rhinos LTD [21]	Proprietary and modified Affero GPL[22]	C#, JavaScript		Yes
Redis		BSD License	ANSI C	Key-value store supporting lists and sets with binary-safe protocol	(unknown)
RethinkDB		GNU APGL for the DBMS, Apache 2 License for the client drivers	C++		(unknown)
Rocket U2	Rocket Software	Proprietary		UniData, UniVerse	Yes (Beta)
Sqrrl Enterprise [23]	sqrrl	Proprietary	Java	Distributed, real-time database featuring cell-level security and massive scalability.	Yes

XML database implementations

Most XML databases are document-oriented databases.

References

[1] http://www.arangodb.org/
[2] http://www.triagens.com/
[3] ArangoDB REST API (http://www.arangodb.org/manuals/current/ImplementorManual.html)
[4] http://basex.org/
[5] https://cloudant.com/
[6] http://www.clusterpoint.com
[7] Clusterpoint DBMS Licensing Options (http://www.clusterpoint.com/licensing/)
[8] Documentation (http://www.couchbase.com/docs/). Couchbase. Retrieved on 2013-09-18.
[9] CouchDB Overview (http://couchdb.apache.org/docs/overview.html)
[10] CouchDB Document API (http://wiki.apache.org/couchdb/HTTP_Document_API)
[11] http://exist-db.org
[12] eXist-db Open Source Native XML Database (http://exist-db.org). Exist-db.org. Retrieved on 2013-09-18.
[13] http://fleetdb.org/
[14] http://fleetdb.org/docs/protocol.html
[15] http://developer.marklogic.com/licensing
[16] MongoDB License (http://www.mongodb.org/display/DOCS/Licensing)
[17] MongoDB REST Interfaces (http://www.mongodb.org/display/DOCS/Http+Interface#HttpInterface-RESTInterfaces)
[18] Extreme Database programming with MUMPS Globals (http://gradvs1.mgateway.com/download/extreme1.pdf)
[19] GTM MUMPS FOSS on SourceForge (http://sourceforge.net/projects/fis-gtm/)
[20] http://www.orientechnologies.com/
[21] http://hibernatingrhinos.com
[22] Ravendb Licensing (http://ravendb.net/licensing)
[23] http://sqrrl.com/

Further reading

€ Assaf Arkin. (2007, September 20). Read Consistency: Dumb Databases, Smart Services. (http://blog.labnotes.org/2007/09/20/read-consistency-dumb-databases-smart-services/) Labnotes:Don,t let the bubble go to your head!

External links

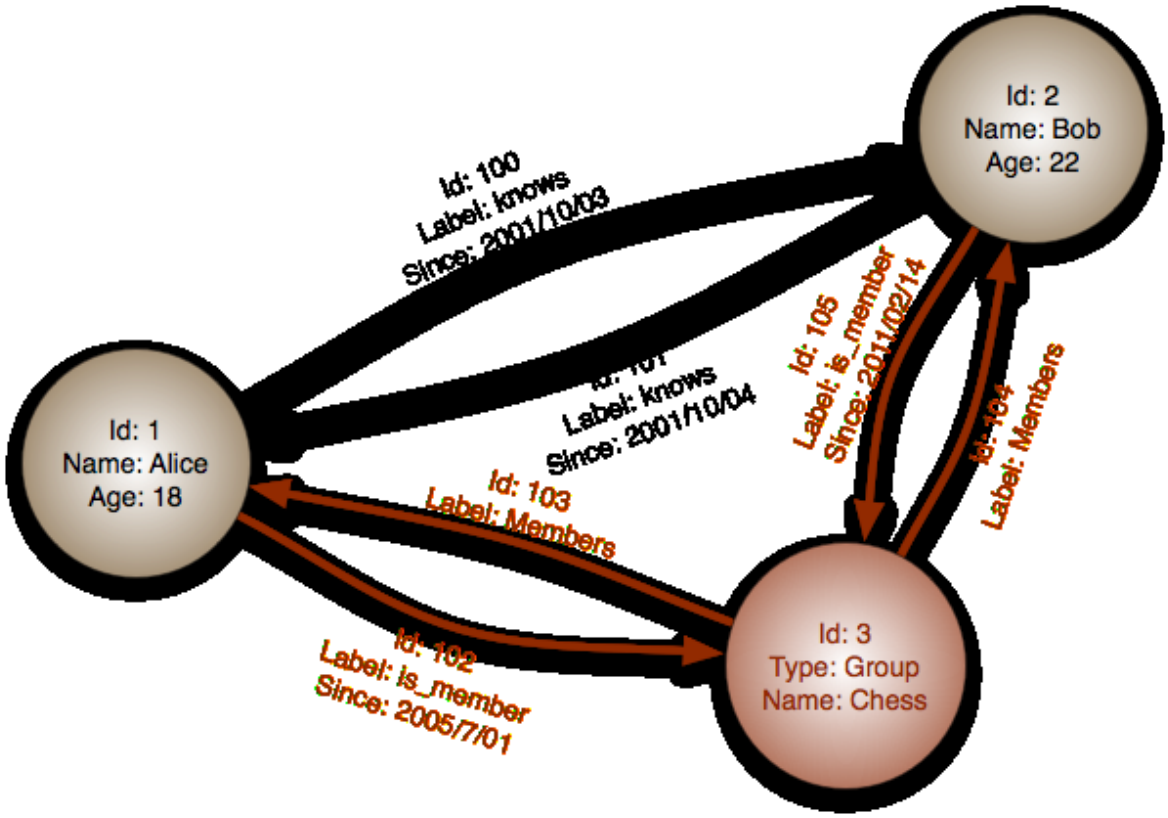
€ http://solprovider.com/articles/20020612&cat=Lotus/IBM

Graph database

A **graph database** is a database that uses graph structures with nodes, edges, and properties to represent and store data. By definition, a graph database is any storage system that provides index-free adjacency. This means that every element contains a direct pointer to its adjacent element and no index lookups are necessary. General graph databases that can store any graph are distinct from specialized graph databases such as triplestores and network databases.

Structure

Graph databases are based on graph theory. Graph databases employ nodes, properties, and edges. Nodes are very similar in nature to the objects that object-oriented programmers will be familiar with.



Nodes represent entities such as people, businesses, accounts, or any other item you might want to keep track of.

Properties are pertinent information that relate to nodes. For instance, if "Wikipedia" were one of the nodes, one might have it tied to properties such as "website", "reference material", or "word that starts with the letter 'w'", depending on which aspects of "Wikipedia" are pertinent to the particular database.

Edges are the lines that connect nodes to nodes or nodes to properties and they represent the relationship between the two. Most of the important information is really stored in the edges. Meaningful patterns emerge when one examines the connections and interconnections of nodes, properties, and edges.

Properties

Compared with relational databases, graph databases are often faster for associative data sets[*citation needed*], and map more directly to the structure of object-oriented applications. They can scale more naturally to large data sets as they do not typically require expensive join operations. As they depend less on a rigid schema, they are more suitable to manage ad hoc and changing data with evolving schemas. Conversely, relational databases are typically faster at performing the same operation on large numbers of data elements.

Graph databases are a powerful tool for graph-like queries, for example computing the shortest path between two nodes in the graph. Other graph-like queries can be performed over a graph database in a natural way (for example graph's diameter computations or community detection).

Graph database projects

The following is a list of several well-known graph database projects:[1]

Name	Version	License	Language	Description
AllegroGraph	4.11 (June 2013)	Proprietary, Clients - Eclipse Public License v1	C, Common Lisp, Java, Python	A RDF and graph database.
ArangoDB [2]	1.3.2 (June 2013)	Apache 2	C, C++ & Javascript	A multi-model document store and graph database.
Bigdata [3]		GPL	Java	A RDF/graph database capable of clustered deployment.
Bitsy [4]	1.5.0	AGPL, Enterprise license (unlimited use, annual/perpetual)	Java	A small, embeddable, durable in-memory graph database
BrightstarDB [5]		MIT License [6]	C#	An embeddable NoSQL database for the .NET platform with code-first data model generation.
DEX[7]	4.8 (2013)	evaluation, research or development use (free) / commercial use	C++	A high-performance and scalable graph database management system from Sparsity Technologies [8], a technology transition company from DAMA-UPC [9]. Its main characteristics is its query performance for the retrieval & exploration of large networks
Filament [10]		BSD	Java	A graph persistence framework and associated toolkits based on a navigational query style.
GraphBase [11]	1.0.03a	Proprietary	Java	A customizable, distributed, small-footprint graph store with a rich tool set from FactNexus [12].
Graphd		Proprietary		The proprietary back-end of Freebase.
Horton [13]		Proprietary	C#	A graph database from Microsoft Research Extreme Computing Group (XCG) [14] based on the cloud programming infrastructure Orleans [15].
HyperGraphDB [16]	1.2 (2012)	LGPL	Java	A graph database supporting generalized hypergraphs where edges can point to other edges.
InfiniteGraph [17]	3.0 (January 2013)	GPLv3	Java	A distributed and cloud-enabled commercial product with flexible licensing.
InfoGrid [18]	2.9.5 (2011)	AGPLv3, free for small entities[19]	Java	A graph database with web front end and configurable storage engines (MySQL, PostgreSQL, Files, Hadoop).

jCoreDB Graph [20]				An extensible database engine with a graph database subproject.
Neo4j	1.9.2 [21] (July 2013)	GPLv3ƒCommunity Edition. Commercial &ƒAGPLv3ƒoptions for Enterprise and Advancedƒeditions[22]	Java	A highly scalable open source graph database that supports ACID, has high-availability clustering for enterprise deployments, and comes with a web-based administration tool that includes full transaction support and visual node-link graph explorer.[23] Neo4j is accessible from most programming languages using its built-in REST web API interface. Neo4j is the most popular graph database in useƒtoday.[24]
OpenLink Virtuoso				A RDF graph database server, deployable as a local embedded instance (as used in the Nepomuk Semantic Desktop), a single-instance network server, or a shared-nothing network cluster instance.
Oracle Spatial and Graph [25]	11.2 (2012)	Proprietary	Java, PL/SQL	1) RDF Semantic Graph: comprehensive W3C RDF graph management in Oracle Database with native reasoning and triple-level label security. 2) Network Data Model property graph: for physical/logical networks with persistent storage and a Java API for in-memory graph analytics.
Oracle NoSQL Database [26]	2.0.39 (2013)	Proprietary	Java	RDF Graph for Oracle NoSQL Database is a feature of Enterprise Edition providing W3C RDF graph capabilities in NoSQL Database.
OrientDB	1.3 (2012)	Apache 2	Java	A document-graph database.
OQGRAPH [27]		GPLv2		A graph computation engine for MySQL, MariaDB and Drizzle.
Ontotext OWLIM [28]	5.3	OWLIM Lite is free OWLIM SE and Enterprise are commercially licenced	Java	A graph database engine, based entirely on Semantic Web standards from W3C: RDF, RDFS, OWL, SPARQL. OWLIM Lite is an "in memory" engine. OWLIM SE is robust standalone database engine. OWLIM Enterprise is a clustered version which offers horizontal scalability and failover support and other enterprise features.
R2DF [29]				R2DF framework for ranked path queries over weighted RDF graphs.
ROIS [30]		Freeware	Modula-2	A programmable knowledge server that supports inheritance and transitivity. Used in OpenGALEN as a Terminology Server.
sones GraphDB		AGPLv3[31]	C#	A graph database and universal access layer (funded by Deutsche Telekom).
Sqrrl Enterprise [23]	v1.1 (2013)	Proprietary	Java	Distributed, real-time graph database featuring cell-level security and massive scalability.
Teradata Aster [32]	v6 (2013)	Proprietary	Java, SQL, Python, C++, R	A high performance, multi-purpose, highly scalable and extensible MPP database incorporating patented engines supporting native SQL, MapReduce and Graph data storage and manipulation. An extensive set of analytical function libraries and data visualization capabilities are also provided.
Titan [33]	0.3 (2013)	Apache 2	Java	A distributed, real-time, transactional graph database developed by Aurelius [34].
VelocityGraph [35]		Open source with proprietary back-end	C#	High performance, scalable & flexible graph database build with VelocityDB [36] object database.
VertexDB [37]		Revised BSD	C	A graph database server that supports automatic garbage collection.

Graph database features

The following table compares the features of the above graph databases.

Name	Graph Model	API	Query Methods	Visualizer	Consistency	Backend	Scalability
AllegroGraph	RDF	Java, Java:Sesame, JavaJena, Python, Ruby, Perl, C#, Clojure, Lisp, Scala, REST	SPARQL 1.1, Prolog, JIG, JavaScript	Gruff - View Graphs, Visual Query Builder for SPARQL and Prolog	ACID	Native Graph Storage	1 Trillion RDF triples
ArangoDB [2]	Property Graph [38]	JavaScript, Blueprints, REST	Graph Traversals via JavaScript, Gremlin		MVCC/ACID	native C/C++	
Bigdata [3]							
Bitsy [4]	Property Graph	Blueprints	Gremlin, Pixy [39]		ACID with optimistic concurrency control	Human-readable JSON-encoded text files with checksums and markers for recovery	
DEX [40]	Labeled and directed attributed multigraph	Java, C++, .NET	Native Java, C# and C++ APIs, Blueprints, Gremlin	Exporting functionality to visualization formats	Consistency, durability and partial isolation and atomicity	Native graph. light and independent data structures with a small memory footprint for storage	Master/Slave replication
Filament [10]							
GraphBase Enterprise(1) [41] GraphBase Agility(2) [42]	(1) mixed, (2) Framework-managed Simple Graph	Java	Bounds Language, embedded java	GraphPad, BoundsPad, Navigator	ACID, graph-based transactions	proprietary native	(1) shared nothing distributed, (2) simple replication, 100+ Billion arcs per server
Graphd							
Horton [13]	Attributed multigraph		Horton Query Language (Regular Language Expression + SQL)			C#, .Net Framework, Asynchronous communication protocols	
HyperGraphDB [16]	Object-oriented multi-relational labeled hypergraph	Custom,Java			MVCC/STM		

InfiniteGraph [17]	Labeled and directed multi-property graph	Java, Blueprints	Java (with parallel, distributed queries), Gremlin	Graph browser for developers. Plugins to allow use of external libraries.	ACID. There is also a parallel, loosely synchronized batch loader.	Objectivity/DB on standard filesystems	Distributed & Sharded. Objectivity/DB was the first DBMS to store a Petabyte of objects.
InfoGrid [18]	Dynamically typed, object-oriented graph, multigraphs, semantic models						
jCoreDB Graph [20]							
Neo4j	Property Graph	Java, Python, JPython, Ruby, JRuby, JavaScript (Node.js), PHP, .NET, Django, Clojure, Spring, Scala, or REST (any language)	Cypher (native/preferred), Native Java APIs (special cases), Traverser API, REST, Blueprints, Gremlin	Data Browser included. Supports a variety of 3rd party tools: Gephi, Linkurio.us, Cytoscape, Tom Sawyer, Keylines, etc.	ACID	Native graph storage with native graph processing engine	Horizontal read scaling via master-slave clustering with cache sharding.
OpenLink							
Oracle Spatial and Graph [25]	RDF graph: Triple & Quad (named graphs); Network Data Model property graph	Java; Apache Jena; PL/SQL	SPARQL 1.1; SPARQL web service end point; SQL	SPARQL-compliant tools; Apache Jena-based tools; XML & JSON-based tools; SQL based tools	ACID	Efficient, compressed, partitioned graph storage; Native persisted in-database inferencing; SPARQL 1.1 & SQL integration; Triple-level label security; Semantic indexing of documents	Parallel load, query, inference; Query controls; Scales from PC to Oracle Exadata; Supports Oracle Real Application Clusters and Oracle Database 8 exabytes
Oracle NoSQL Database [26]	RDF graph: Triple default graph, Triple & Quad named graphs	Java (Apache Jena)	SPARQL 1.1; SPARQL web service end point	SPARQL-compliant tools; Apache Jena-based tools; XML & JSON-based tools	ACID; Configurable consistency & durability policies	Key/value store; W3C SPARQL 1.1 & Update; In-memory RDFS/OWL inferencing	Parallel load/query; Query controls for: parallel execution, timeout, query optimization hints

OrientDB	Property Graph	Java Traverser API, Blueprints, Rexster	Own SQL-like Query Language, Gremlin		ACID, MVCC	Custom on disc or in memory	
OQGRAPH [27]							
R2DF [29]							
ROIS [30]							
sones GraphDB							
Sqrrl Enterprise [43]	Property Graph	Thrift, Blueprint	Gremlin	Integrates with 3rd party tools	Eventually Consistent	Accumulo	Distributed cluster with tens of trillions of edges [44]
Titan [45]	Property Graph	Java, Blueprints, REST, RexPro binary protocol (any language)	Gremlin, SPARQL	Integrates with 3rd party tools	ACID or Eventually Consistent	Cassandra, HBase, Berkeley DB	Distributed cluster (120 billion+ edges) or single server.
VertexDB [37]							

Distributed Graph Processing

- Angrapa [46] - graph package in Hama [47], a bulk synchronous parallel (BSP) platform
- Apache Hama [47] - a pure BSP(Bulk Synchronous Parallel) computing framework on top of HDFS (Hadoop Distributed File System) for massive scientific computations such as matrix, graph and network algorithms.
- Bigdata [3] - a RDF/graph database capable of clustered deployment.
- Faunus [48] - a Hadoop-based graph computing framework that uses Gremlin as its query language. Faunus provides connectivity to Titan, Rexster-fronted graph databases, and to text/binary graph formats stored in HDFS. Faunus is developed by Aurelius [34].
- FlockDB - an open source distributed, fault-tolerant graph database based on MySQL and the Gizzard framework for managing Twitter-like graph data (single-hop relationships) FlockDB on GitHub [49].
- Giraph [50] - a Graph processing infrastructure that runs on Hadoop (see Pregel).
- GraphBase [51] - Enterprise Edition supports embedding of callable Java Agents within the vertices of a distributed graph.
- GoldenOrb [52] - Pregel implementation built on top of Apache Hadoop
- GraphLab [53] - A framework for machine learning and data mining in the cloud
- HipG [54] - a library for high-level parallel processing of large-scale graphs. HipG is implemented in Java and is designed for distributed-memory machine
- InfiniteGraph [17] - a commercially available distributed graph database that supports parallel load and parallel queries.
- JPregel [55] - In-memory java based Pregel implementation
- KDT [56] - An open-source distributed graph library with a Python front-end and C++/MPI backend (Combinatorial BLAS [57]).

- € OpenLink Virtuoso - the shared-nothing Cluster Edition supports distributed graph data processing.
- € Oracle Spatial and Graph [25] - loading, inferencing, and querying workloads are automatically and transparently distributed across the nodes in an Oracle Real Application Cluster, Oracle Exadata Database Machine, and Oracle Database Appliance.
- € Phoebus [58] - Pregel implementation written in Erlang
- € Pregel [59] - Google's internal graph processing platform, released details in ACM paper.
- € Powergraph [60] - Distributed graph-parallel computation on natural graphs.
- € Sedge [61] - A framework for distributed large graph processing and graph partition management (including an open source version of Google's Pregel)
- € Signal/Collect [62] - a framework for parallel graph processing written in Scala
- € Sqrrl Enterprise - distributed graph processing utilizing Apache Accumulo and featuring cell-level security, massive scalability, and JSON support
- € Titan [45] - A distributed, disk-based graph database developed by Aurelius [34].
- € Trinity [63] - Distributed in-memory graph engine under development at Microsoft Research Labs.
- € Parallel Boost Graph Library (PBGL) [64] - a C++ library for graph processing on distributed machines, part of Boost framework.
- € Mizan [65] - An optimized Pregel clone that can be deployed easily on Amazon EC2, or local clusters, or stand-alone Linux systems.

APIs and Graph Query/Programming Languages

- € Bounds Language [66] - terse C-style syntax which initiates concurrent traversals in GraphBase and supports interaction between them.
- € Blueprints [67] - a Java API for Property Graphs from TinkerPop [68] and supported by a few graph database vendors.
- € Blueprints.NET [69] - a C#/.NET API for generic Property Graphs.
- € Bulbflow [70] - a Python persistence framework for Rexster, Titan, and Neo4j Server.
- € Cypher [71] - a declarative graph query language for ƒNeo4j that enables ad hoc as well as programmatic (SQL-like) access to the graph
- € Gremlin [72] - an open-source graph programming language that works over various graph database systems.
- € Neo4jClient [73] ƒ- a .NET client for accessing Neo4j.
- € Neography [74] ƒ- a thin Ruby wrapper that provides access to Neo4j via REST.
- € Neo4jPHP [75] - a PHP library wrapping the Neo4j graph database.
- € NodeNeo4j [76] ƒ- a Node.js driver for Neo4j that provides access to Neo4j via REST
- € Pacer [77] - a Ruby dialect/implementation of the Gremlin graph traversal language.
- € Pipes [78] - a lazy dataflow framework written in Java that forms the foundation for various property graph traversal languages.
- € Pixy [39] - a declarative graph query language that works on any Blueprints-compatible graph database
- € PYBlueprints [79] - a Python API for Property Graphs.
- € Pygr [80] - a Python API for large-scale analysis of biological sequences and genomes, with alignments represented as graphs.
- € Rexster [81] - a graph database server that provides a REST or binary protocol API (RexPro). Supports Titan, Neo4j, OrientDB, Dex, and any TinkerPop/Blueprints-enabled graph.
- € SPARQL - a query language for databases, able to retrieve and manipulate data stored in Resource Description Framework format.
- € SPASQL [82] - an extension of the SQL standard, allowing execution of SPARQL queries within SQL statements, typically by treating them as subquery or function clauses. This also allows SPARQL queries to be issued through "traditional" data access APIs (ODBC, JDBC, OLE DB, ADO.NET, etc.)

- Spring Data Neo4j [83]- an extension to Spring Data [84] (part of the Spring Framework), providing direct/native access to Neo4j
- Oracle SQL and PL/SQL APIs [25] - have graph extensions for Oracle Spatial and Graph.
- Styx [85] (previously named Pipes.Net) - a data flow framework for C#/.NET for processing generic graphs and Property Graphs.
- Thunderdome [86] - a Titan Rexster Object-Graph Mapper for Python

References

[1] http://graph-database.org
[2] http://www.arangodb.org
[3] http://www.bigdata.com/blog
[4] http://bitbucket.org/lambdazen/bitsy
[5] http://www.brightstardb.com
[6] http://brightstardb.com/blog/2013/02/brightstardb-goes-open-source/
[7] http://sparsity-technologies.com/dex
[8] http://sparsity-technologies.com
[9] http://www.dama.upc.edu/technology-transfer/dex
[10] http://filament.sourceforge.net/
[11] http://graphbase.net/
[12] http://factnexus.com/
[13] http://research.microsoft.com/en-us/projects/ldg
[14] http://research.microsoft.com/en-us/labs/xcg
[15] http://research.microsoft.com/en-us/projects/orleans/default.aspx
[16] http://www.hypergraphdb.org
[17] http://infinitegraph.com
[18] http://infogrid.org/
[19] http://infogrid.org/wiki/Docs/License
[20] http://www.jcoredb.org
[21] http://www.neo4j.org/download
[22] neo4j.org (http://www.neo4j.org)
[23] Neo4j, World,s Leading Graph Database (http://www.neotechnology.com/neo4j-graph-database/). Retrieved September 16, 2013.
[24] DB-Engines Ranking of Graph DBMS (http://db-engines.com/en/ranking/graph+dbms). Retrieved July 19, 2013.
[25] http://www.oracle.com/technetwork/database-options/spatialandgraph/overview/index.html
[26] http://www.oracle.com/technetwork/products/nosqldb/overview/index.html
[27] http://openquery.com/graph
[28] http://www.ontotext.com/owlim
[29] http://dl.acm.org/citation.cfm?id=1988736/
[30] http://rois.eggbird.eu/
[31] http://sones.com/
[32] http://www.asterdata.com/
[33] http://titan.thinkaurelius.com/
[34] http://thinkaurelius.com
[35] http://www.VelocityGraph.com
[36] http://www.VelocityDB.com
[37] http://www.dekorte.com/projects/opensource/vertexdb/
[38] https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model
[39] https://github.com/lambdazen/pixy/wiki
[40] http://sparsity-technologies.com/dex
[41] http://graphbase.net/Enterprise.html/
[42] http://graphbase.net/Agility.html/
[43] http://sqrrl.com
[44] http://www.pdl.cmu.edu/SDI/2013/slides/big_graph_nsa_rd_2013_56002v1.pdf
[45] http://thinkaurelius.github.com/titan/
[46] http://wiki.apache.org/hama/GraphPackage
[47] http://incubator.apache.org/hama/
[48] http://thinkaurelius.github.com/faunus/
[49] https://github.com/twitter/flockdb

[50] http://incubator.apache.org/giraph/
[51] http://graphbase.net/Enterprise.html
[52] http://www.goldenorbos.org
[53] http://graphlab.org
[54] http://www.cs.vu.nl/~ekr/hipg/
[55] http://kowshik.github.com/JPregel/
[56] http://kdt.sourceforge.net
[57] http://gauss.cs.ucsb.edu/~aydin/CombBLAS/html/index.html
[58] http://github.com/xslogic/phoebus
[59] http://portal.acm.org/citation.cfm?id=1582723
[60] http://graphlab.org/powergraph-presented-at-osdi/
[61] http://grafia.cs.ucsb.edu/sedge/
[62] http://code.google.com/p/signal-collect/
[63] http://research.microsoft.com/en-us/projects/trinity/
[64] http://www.boost.org/doc/libs/1_51_0/libs/graph_parallel/doc/html/index.html
[65] http://thegraphsblog.wordpress.com/the-graph-blog/mizan/
[66] http://graphbase.net/JavaAPIHelp.html#BoundsLanguage
[67] http://blueprints.tinkerpop.com
[68] http://www.tinkerpop.com/
[69] https://github.com/Vanaheimr/Blueprints.NET
[70] http://bulbflow.com
[71] http://docs.neo4j.org/chunked/snapshot/cypher-query-lang.html
[72] http://gremlin.tinkerpop.com/
[73] http://hg.readify.net/neo4jclient
[74] https://github.com/maxdemarzi/neography/
[75] https://github.com/jadell/neo4jphp/wiki
[76] https://github.com/thingdom/node-neo4j
[77] http://github.com/pangloss/pacer
[78] http://pipes.tinkerpop.com
[79] http://pypi.python.org/pypi/pyblueprints/0.1
[80] http://code.google.com/p/pygr/
[81] http://rexster.tinkerpop.com
[82] http://www.w3.org/wiki/SPASQL
[83] http://www.springsource.org/spring-data/neo4j
[84] http://www.springsource.org/spring-data
[85] https://github.com/ahzf/Styx
[86] https://github.com/StartTheShift/thunderdome

External links

- NoSQL Frankfurt 2010 - The GraphDB Landscape and sones (http://www.slideshare.net/ahzf/nosql-frankfurt-2010-the-graphdb-landscape-and-sones)
- Graph Databases and the Future of Large-Scale Knowledge Management (http://highscalability.com/paper-graph-databases-and-future-large-scale-knowledge-management)
- Graphs in the database: SQL meets social networks (http://techportal.ibuildings.com/2009/09/07/graphs-in-the-database-sql-meets-social-networks/)
- Social networks in the database: using a graph database (http://blog.neo4j.org/2009/09/social-networks-in-database-using-graph.html)
- Scaling Online Social Networks without Pains (http://netdb09.cis.upenn.edu/netdb09papers/netdb09-final3.pdf)
- Large-scale Graph Computing at Google (http://googleresearch.blogspot.com/2009/06/large-scale-graph-computing-at-google.html)
- Eric Lai. (2009, July 1). No to SQL? Anti-database movement gains steam (http://www.computerworld.com/s/article/9135086/No_to_SQL_Anti_database_movement_gains_steam_)

€ Renzo Angles, Claudio Gutierrez. Survey of graph database models (http://portal.acm.org/citation.cfm?id=1322433). ACM Computing Surveys, Feb. 2008.
€ InfoGrid (http://infogrid.org/) - an open-source application platform including a graph database
€ Rodriguez, M.A., Neubauer, P, The Graph Traversal Pattern (http://arxiv.org/abs/1004.1001) article.
€ Optimizing Schema-Last Tuple-Store Queries in Graphd (http://portal.acm.org/citation.cfm?id=1807283) SIGMOD 2010

NoSQL

A **NoSQL** database provides a mechanism for storage and retrieval of data that employs less constrained consistency models than traditional relational databases. Motivations for this approach include simplicity of design, horizontal scaling and finer control over availability. NoSQL databases are often highly optimized key€value stores intended for simple retrieval and appending operations, with the goal being significant performance benefits in terms of latency and throughput. NoSQL databases are finding significant and growing industry use in big data and real-time web applications. NoSQL systems are also referred to as "Not only SQL" to emphasize that they do in fact allow SQL-like query languages to be used.

History

Carlo Strozzi used the term *NoSQL* in 1998 to name his lightweight, open-source relational database that did not expose the standard SQL interface. Strozzi suggests that, as the current NoSQL movement "departs from the relational model altogether; it should therefore have been called more appropriately 'NoREL'.

Eric Evans (then a Rackspace employee) reintroduced the term *NoSQL* in early 2009 when Johan Oskarsson of Last.fm wanted to organize an event to discuss open-source distributed databases. The name attempted to label the emergence of a growing number of non-relational, distributed data stores that often did not attempt to provide atomicity, consistency, isolation and durability guarantees that are key attributes of classic relational database systems.

Taxonomy

There have been various approaches to classify NoSQL databases, each with different categories and subcategories. Because of the variety of approaches and overlaps it is difficult to get and maintain an overview of non-relational databases. Nevertheless, the basic classification that most would agree on is based on data model. A few of these and their prototypes are:

€ **Column**: HBase, Accumulo
€ **Document**: MongoDB, Couchbase
€ **Key-value** : Dynamo, Riak, Redis, Cache, Project Voldemort
€ **Graph**: Neo4J, Allegro, Virtuoso

Classification based on data model

Stephen Yen in his blog post "NoSQL is a Horseless Carriage" suggests the following:[1]

Term	Matching Database
KV Store	Keyspace, Flare, SchemaFree, RAMCloud, Oracle NoSQL Database (OnDB)
KV Store - Eventually consistent	Dynamo, Voldemort, Dynomite, SubRecord, Mo8onDb, DovetailDB
KV Store - Hierarchical	GT.m, Cache
KV Store - Ordered	TokyoTyrant, Lightcloud, NMDB, Luxio, MemcacheDB, Actord
KV Cache	Memcached, Repcached, Coherence, Hazelcast, Infinispan, EXtremeScale, JBossCache, Velocity, Terracoqua
Tuple Store	Gigaspaces, Coord, Apache River
Object Database	ZopeDB, DB40, Shoal
Document Store	CouchDB, Cloudant, Couchbase, MongoDB, Jackrabbit, XML-Databases, ThruDB, CloudKit, Prsevere, Riak-Basho, Scalaris
Wide Columnar Store	BigTable, HBase, Apache Cassandra, Hypertable, KAI, OpenNeptune, Qbase, KDI

Classification based on feature

Ben Scofield categorized NoSQL databases based on nonfunctional categories (•(il)ities•) plus a rating of their feature coverage: [*citation needed*]

Data Model	Performance	Scalability	Flexibility	Complexity	Functionality
Key€value Stores	high	high	high	none	variable (none)
Column Store	high	high	moderate	low	minimal
Document Store	high	variable (high)	high	low	variable (low)
Graph Database	variable	variable	high	high	graph theory
Relational Database	variable	variable	low	moderate	relational algebra.

Examples

Document store

The central concept of a document store is the notion of a "document". While each document-oriented database implementation differs on the details of this definition, in general, they all assume that documents encapsulate and encode data (or information) in some standard formats or encodings. Encodings in use include XML, YAML, and JSON as well as binary forms like BSON, PDF and Microsoft Office documents (MS Word, Excel, and so on).

Different implementations offer different ways of organizing and/or grouping documents:

- Collections
- Tags
- Non-visible Metadata
- Directory hierarchies

Compared to relational databases, for example, collections could be considered as tables as well as documents could be considered as records. But they are different: every record in a table has the same sequence of fields, while documents in a collection may have fields that are completely different.

Documents are addressed in the database via a unique **key** that represents that document. One of the other defining characteristics of a document-oriented database is that, beyond the simple key-document (or key€value) lookup that you can use to retrieve a document, the database will offer an API or query language that will allow retrieval of

documents based on their contents. Some NoSQL document stores offer an alternative way to retrieve information using MapReduce techniques, in CouchDB the usage of MapReduce is mandatory if you want to retrieve documents based on the contents, this is called "Views" and it's an indexed collection with the results of the MapReduce algorithms.

Name	Language	Notes
BaseX	Java, XQuery	XML database
Cloudant	Erlang, Java, Scala, C	JSON store (online service)
Clusterpoint	C++	XML, geared for Full text search
Couchbase Server	Erlang, C, C++	Support for JSON and binary documents
Apache CouchDB	Erlang	JSON database
djondb[2][3][4]	C++	JSON, ACID Document Store
ElasticSearch	Java	JSON, Search engine
eXist	Java, XQuery	XML database
Jackrabbit	Java	Java Content Repository implementation
IBM Lotus Notes and Lotus Domino	LotusScript, Java, IBM X Pages, others	MultiValue
MarkLogic Server	XQuery, Java, REST	XML database with support for JSON, text, and binaries
MongoDB	C++, C#, Go	BSON store (binary format JSON)
Oracle NoSQL Database	Java, C	
OrientDB	Java	JSON, SQL support
CoreFoundation Property list	C, C++, Objective-C	JSON, XML, binary
Sedna	XQuery, C++	XML database
SimpleDB	Erlang	online service
TokuMX	C++, C#, Go	MongoDB with Fractal Tree indexing
OpenLink Virtuoso	C++, C#, Java, SPARQL	middleware and database engine hybrid

Graph

This kind of database is designed for data whose relations are well represented as a graph (elements interconnected with an undetermined number of relations between them). The kind of data could be social relations, public transport links, road maps or network topologies, for example.

Name	Language	Notes
AllegroGraph	SPARQL	RDF GraphStore
IBM DB2	SPARQL	RDF GraphStore added in DB2 10
DEX	Java, C++, .NET	High-performance graph database
FlockDB	Scala	
InfiniteGraph	Java	High-performance, scalable, distributed graph database
Neo4j	Java	
OpenLink Virtuoso	C++, C#, Java, SPARQL	middleware and database engine hybrid
OrientDB	Java	
Sones GraphDB	C#	

Sqrrl Enterprise	Java	Distributed, real-time graph database featuring cell-level security
OWLIM	Java, SPARQL 1.1	RDF graph store with reasoning
VelocityGraph [5]	C#	Fully Tinkerpop Blueprints [6] compliant. Scalable hybrid object database and graph database

Key‡value stores

Key€value stores allow the application to store its data in a schema-less way. The data could be stored in a datatype of a programming language or an object. Because of this, there is no need for a fixed data model. The following types exist:

KV - eventually consistent

- Apache Cassandra
- Dynamo
- Hibari
- OpenLink Virtuoso
- Project Voldemort
- Riak

KV - hierarchical

- GT.M
- InterSystems Cach…

KV - cache in RAM

- memcached
- OpenLink Virtuoso
- Hazelcast
- Oracle Coherence

KV - solid state or rotating disk

- Aerospike
- BigTable
- CDB
- Couchbase Server
- Keyspace
- LevelDB
- MemcacheDB (using Berkeley DB)
- MongoDB
- OpenLink Virtuoso
- phpFastCache
- Tarantool
- Tokyo Cabinet
- Tuple space
- Oracle NoSQL Database
- IBM WebSphere DataPower XC10 Appliance

KV - ordered

- Berkeley DB
- FoundationDB
- IBM Informix C-ISAM
- InfinityDB
- MemcacheDB
- NDBM

Object database

- db4o
- GemStone/S
- InterSystems Cach…
- JADE
- NeoDatis ODB
- ObjectDB
- Objectivity/DB
- ObjectStore
- ODABA
- OpenLink Virtuoso
- Versant Object Database
- WakandaDB
- ZODB

Tabular

- Apache Accumulo
- BigTable
- Apache Hbase
- Hypertable
- Mnesia
- OpenLink Virtuoso

Tuple store

- Apache River
- OpenLink Virtuoso
- Tarantool

Triple/Quad Store (RDF) database

- Meronymy SPARQL Database Server
- Virtuoso Universal Server
- Ontotext-OWLIM
- Apache JENA
- Oracle NoSQL database

Hosted

- Freebase
- OpenLink Virtuoso
- Datastore on Google Appengine
- Amazon DynamoDB
- Cloudant Data Layer (CouchDB)

Multivalue databases

- Northgate Information Solutions Reality, the original Pick/MV Database
- Extensible Storage Engine (ESE/NT)
- OpenQM
- Revelation Software's OpenInsight
- Rocket U2
- D3 Pick database
- InterSystems Cach…
- InfinityDB

Cell database

- [] Boardwalk

References

[1] A Yes for a NoSQL Taxonomy (http://highscalability.com/blog/2009/11/5/a-yes-for-a-nosql-taxonomy.html). High Scalability (2009-11-05). Retrieved on 2013-09-18.

[2] The enterprise class NoSQL database (http://djondb.com). djondb. Retrieved on 2013-09-18.

[3] http://tinman.cs.gsu.edu/~raj/8711/sp13/djondb/Report.pdf

[4] Undefined Blog: Meeting with DjonDB (http://undefvoid.blogspot.com/2013/03/meeting-with-djondb.html). Undefvoid.blogspot.com. Retrieved on 2013-09-18.

[5] https://github.com/VelocityDB/VelocityGraph

[6] https://github.com/Loupi/Frontenac

Further reading

- Pramod Sadalage and Martin Fowler (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley. ISBN 0-321-82662-0.
- Christof Strauch (2012). "NoSQL Databases" (http://www.christof-strauch.de/nosqldbs.pdf).
- Moniruzzaman AB, Hossain SA (2013). "NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison" (http://arxiv.org/abs/1307.0191).
- Kai Orend (2013). *Analysis and Classification of NoSQL Databases and Evaluation of their Ability to Replace an Object-relational Persistence Layer* (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.184.483&rep=rep1&type=pdf).
- Ganesh Krishnan, Sarang Kulkarni, Dharmesh Kirit Dadbhawala. "Method and system for versioned sharing, consolidating and reporting information" (https://www.google.com/patents/US7383272?pg=PA1&dq=ganesh+krishnan&hl=en&sa=X).

External links

- Christoph Strauch. "NoSQL whitepaper" (http://www.christof-strauch.de/nosqldbs.pdf). Hochschule der Medien, Stuttgart.
- Martin Fowler. "NoSQL Guide" (http://martinfowler.com/nosql.html).
- Stefan Edlich. "NoSQL database List" (http://nosql-database.org/).
- Peter Neubauer (2010). "Graph Databases, NOSQL and Neo4j" (http://www.infoq.com/articles/graph-nosql-neo4j).
- Sergey Bushik (2012). "A vendor-independent comparison of NoSQL databases: Cassandra, HBase, MongoDB, Riak" (http://www.networkworld.com/news/tech/2012/102212-nosql-263595.html). NetworkWorld.

NewSQL

NewSQL is a class of modern relational database management systems that seek to provide the same scalable performance of NoSQL systems for online transaction processing (read-write) workloads while still maintaining the ACID guarantees of a traditional database system.

History

The term was first used by 451 Group analyst Matthew Aslett in a 2011 research paper discussing the rise of new database systems as challengers to established vendors. Many enterprise systems that handle high-profile data (e.g., financial and order processing systems) also need to be able to scale but are unable to use NoSQL solutions because they cannot give up strong transactional and consistency requirements. The only options previously available for these organizations were to either purchase a more powerful single-node machine or develop custom middleware that distributes queries over traditional DBMS nodes. Both approaches are prohibitively expensive and thus are not an option for many. Thus, in this paper, Aslett discusses how NewSQL upstarts are poised to challenge the supremacy of commercial vendors, in particular Oracle.

Systems

Although NewSQL systems vary greatly in their internal architectures, the two distinguishing features common amongst them is that they all support the relational data model and use SQL as their primary interface. One of the first known NewSQL systems is the H-Store parallel database system.

NewSQL systems can be loosely grouped into three categories:

New architectures

The first type of NewSQL systems are completely new database platforms. These are designed to operate in a distributed cluster of shared-nothing nodes, in which each node owns a subset of the data. Though many of the new databases have taken different design approaches, there are two primary categories evolving. The first type of system sends the execution of transactions and queries to the nodes that contain the needed data. SQL queries are split into query fragments and sent to the nodes that own the data. These databases are able to scale linearly as additional nodes are added.

- General-purpose databases ƒ These maintain the full functionality of traditional databases, handling all types of queries. These databases are often written from scratch with a distributed architecture in mind, and include components such as distributed concurrency control, flow control, and distributed query processing. This includes Google Spanner, Clustrix, NuoDB and TransLattice.

- In-memory databases – The applications targeted by these NewSQL systems are characterized as having a large number of transactions that (1) are short-lived (i.e., no user stalls), (2) touch a small subset of data using index lookups (i.e., no full table scans or large distributed joins), and (3) are repetitive (i.e., executing the same queries with different inputs). These NewSQL systems achieve high performance and scalability by eschewing much of the legacy architecture of the original System R design, such as heavyweight recovery or concurrency control algorithms. Two example systems in this category are VoltDB and GoPivotal's SQLFire.

MySQL Engines

The second category are highly optimized storage engines for SQL. These systems provide the same programming interface as MySQL, but scale better than built-in engines, such as InnoDB. Examples of these new storage engines include TokuDB, MemSQL, and Akiban.

Transparent sharding

These systems provide a sharding middleware layer to automatically split databases across multiple nodes. Examples of this type of system includes dbShards, Scalearc, ScaleBase and MySQL Cluster.

References

Article Sources and Contributors

Database *Source*: http://en.wikipedia.org/w/index.php?oldid=577356344 *Contributors*: *drew, 05winsjp, 069952497a, 10285658sdsaa, 10metreh, 110808028 amol, 16@r, 2001:558:6033:AE:4A:74EE:6356:A9B8, 206.31.111.xxx, 25or6to4, 28421u2232nfenfcenc, 28nebraska, 2D, 4twenty42o, 65.10.163.xxx, APH, Aaron Brenneman, Abhikumar1995, Addihockey10, Aditya gopal3, Admfirepanther, Adrian J. Hunter, Aepanico, Afluegel, Ahodgkinson, Ahoerstemeier, Ahy1, Aitias, Aj.robin, Akamad, Al Wiseman, Alain Amiouni, Alansohn, Alasdair, Ale jrb, Allan McInnes, Allecher, Alpha Quadrant (alt), Alphax, Alzpp, Amaraiel, Amaury, Amd628, Anders Torlind, Andonic, Andre Engels, Andrewferrier, AndriuZ, Angela, Anikingos, AnjaliSinha, AnmaFinotera, Ann Stouter, AnonUser, Anonymous Dissident, Antandrus, Antrax, Apparition11, Arbitrarily0, Arcann, Arctic Kangaroo, Argon233, Arjun01, Armen1304, ArnoLagrange, Arthena, Arthur Rubin, Arved, ArwinJ, AsceticRose, Asyndeton, AtheWeatherman, Atkinsdc, Autumn Wind, AutumnSnow, Avenged Eightfold, AwamerT, Ayecee, AzaToth, Baa, Babbling.Brook, Barneca, Bbatsell, Bbb23, Bblank, Bcartolo, Bcontins, Beeblebrox, Beetstra, Beland, Ben Ben, Ben-Zin, Benni39, BentlijDB, Bentogoa, Bernd in Japan, Beta M, Betterusername, Bharath357, BigPimpinBrah, Bjcubsfan, Bkhouser, Blanchardb, BlindEagle, Blood Red Sandman, BluCreator, Bluemask, Bluerocket, BobStepno, Bobblewik, Bogdangiusca, Bogey97, Boing! said Zebedee, Boli1107, Bongwarrior, Bowlderizer, Branzman, Brick Thrower, BrokenSphere, BryanG, Btilm, Bubba hotep, Bunnyhop11, Burner0718, Buzzimu, Bwhynot14, C12H22O11, CIreland, COMPFUNK2, CableCat, Calabe1992, Call me Bubba, Callanecc, Calliopejen1, Caltas, Calutuigor, Cambalachero, Cambapp, Cammo33, Camw, Can't sleep, clown will eat me, CanisRufus, Canterbury Tail, Cantras, Capricorn42, Captain-n00dle, Captain-tucker, Carbonite, CardinalDan, Caster23, CasualVisitor, Cavanagh, Cenarium, CesarB, Cevalsi, Ceyjan, Chaojoker, Cheolsoo, Chester Markel, Childzy, Chirpy, Chocolateboy, ChorizoLasagna, Chrax, Chris 73, Chris G, ChrisGualtieri, Chrislk02, Chrism, Christophe.billiottet, Chriswiki, Chtuw, Chuckhoffmann, Chuunen Baka, Clarince63, Clark89, Click23, Closedmouth, Colindolly, Colonies Chris, Cometstyles, Commander Keane, Compfreak7, Comps, Constructive, Conversion script, Courcelles, Cpereyra, Cpl Syx, Cpuwhiz11, Craftyminion, Craig Stuntz, Crashdoom, Credema, Crucis, Cryptic, Culverin, Cyan, Cybercobra, Cyberjoac, CynicalMe, D. Recorder, DARTH SIDIOUS 2, DEddy, DFS454, DJ Clayworth, DVD R W, DVdm, DamnRandall, Dan100, Dancayta, Dancter, Danhash, Daniel.Cardenas, DanielCD, Danieljamesscott, Danim, Dart88, Darth Mike, Darth Panda, Darthvader023, Davewild, David Fuchs, David0811, Dbates1999, Dbfirs, DePiep, Dead3y3, DeadEyeArrow, DeadlyAssassin, Deathlasersonline, Decrease789, DeirdreGerhardt, Denisarona, DerBorg, DerHexer, Deville, Dgw, Diamondland, DigitalEnthusiast, Discospinster, Djordjes, Djsasso, Dkastner, Dkwebsub, Doc glasgow, Doddsy1993, Dogposter, Donama, Doniago, Donner60, DougBarry, Dougofborg, Doulos Christos, DragonLord, Dreadstar, Dreamyshade, Drivenapart, Drumroll99, Duyanfang, Dwolt, Dysepsion, E23, Eagleal, Earlypsychosis, EarthPerson, EastTN, Echartre, Eddiecarter1, Eddiejcarter, Edgar181, Edgarde, Edivorce, Edward, Eeekster, Ejrrjs, ElKevbo, Elwikipedista, Epbr123, Era7bd, Eric Bekins, Eric Burnett, Ericlaw02, Erikrj, Escape Orbit, Etxrge, EugeneZelenko, EvergreenFir, Everyking, Evildeathmath, Excirial, Exor674, Explicit, Eyesnore, Ezeu, FFGeyer, Fang Aili, FatalError, Favonian, Feedmecereal, FetchcommsAWB, Feydey, Fieldday-sunday, Filx, Finlay McWalter, Flewis, Flubeca, Fluffernutter, Flyer22, FlyingToaster, Fooker69, Fortdj33, Foxfax555, Fraggle81, Frankman, Franky21, Franl, Fratrep, Freebiekr, Frsparrow, Fubar Obfusco, Furrykef, Fuzzie, F", G12kid, GDonato, GHe, GLaDOS, Gadfium, Gail, Galzigler, Garyzx, Gburd, Giftlite, Ginsengbomb, Ginsuloft, Girl2k, Gishac, Glacialfox, GlenPeterson, GnuDoyng, Gogo Dodo, GoingBatty, Gonfus, Gozzy345, Graeme Bartlett, GraemeL, Graham87, GrayFullbuster, GregWPhoto, Gregfitzy, GregorB, Grim23, Grsmca, Grstain, Gsallis, Gscshoyru, Gwizard, Gzkn, Haakon, Hadal, HaeB, Hamtechperson, Hankhuck, HappyInGeneral, Harej, Hasek is the best, HeliXx, Helixblue, Helloher, HexaChord, Heymid, Heysim0n, Hotstaff, Hshoemark, Hugsandy, Huntthetroll, Hurricane111, HybridBoy, Hydrogen Iodide, IComputerSaysNo, IElonex!, Iced Kola, Igoldste, Imfargo, Imnotminkus, Imran, Informatwr, Insineratehymn, Inspector 34, Intgr, IrfanSha, Ironman5247, Isfisk, Itafran2010, ItsZippy, Ixfd64, J.delanoy, JCLately, JForget, JJdaboss, JLaTondre, JMRyan, Ja 62, JaGa, Jab843, Jabby11, Jack Greenmaven, Jackacon, JamesBWatson, Jan1nad, Jarble, Jasimab, Jasper Deng, Jauerback, Javert, Jaxl, Jay, Jb-adder, Jclemens, Jdlambert, JeffTan, JeffreyYasskin, Jennavecia, JephapE, Jerome Charles Potts, Jk2q3jrklse, Jmanigold, Jni, JoanneB, Joel7687, Joffeloff, John Vandenberg, John of Reading, Johnuniq, Jojalozzo, Jonathan Webley, JonathanFreed, Jondel, Jonearles, Jonwynne, Joshnpowell, Joshwa1234567890, Journalist, Jschnur, Jstaniek, JunWan, Jvhertum, Jwoodger, Jwy, KILLERKEA23, Kanonkas, Karlhahn, Karmafist, Katalaveno, Keenan Pepper, Keilana, Kekekecakes, Kellyk99, Kenny sh, Kevins, KeyStroke, Khazar2, Khoikhoi, Kiand, Kimberly ayoma, Kimera Kat, King of Hearts, Kingius, Kingpin13, KingsleyIdehen, Kivar2, Kkailas, Knbanker, Koavf, Kocio, Komal.Ar, KoshVorlon, KotetsuKat, Kozmando, Kraftlos, Krashlandon, Kslays, Kukini, Kunaldeo, Kungfuadam, Kuru, Kushal one, Kvasilev, Kwiki, KyraVixen, Kzzl, L Kensington, LC, Lamp90, LaosLos, Larsinio, Latka, Leaderofearth, Leandrod, LeaveSleaves, LeeHam2007, LeonnicholIs07, LessHeard vanU, Levin, Levin Carsten, Lexo, Lflores92201, Lfstevens, Lguzenda, Lights, LindaEllen, Lingliu07, Lingwitt, Linkspamremover, LittleOldMe, LittleWink, Llyntegid, Lod, Logan, Lotje, Lovefamosos, Lovelac7, Lowellian, Lradrama, Lsschwar, LuK3, Lucyin, Lugia2453, Luizfsc, Luna Santin, M.badnjki, M4gnum0n, MBisanz, MECiAf., MER-C, MJunkCat, Machdohvah, Madhava 1947, Magioladitis, Majorly, Makecat, Malvikiran, Mandarax, Manikandan 2030, Mannafredo, Marasmusine, Mark Arsten, Mark Renier, MarkSutton, MartinSpamer, Materialscientist, Mathewforyou, Mato, Matthewrbowker, Matticus78, Mattisse, Maty18, Maury Markowitz, Max Naylor, Maxferrario, Maxime.Debosschere, Maxmarengo, Mayur, Mazca, Mboverload, McGeddon, Mdd, Meaghan, Mean as custard, Mediran, Megatronium, Melody Lavender, Melucky2getu, Mentifisto, Menublogger, Mercy11, Methnor, Mhkay, Michael Hardy, Michael Slone, Microchip08, Mike Dillon, Mike Rosoft, Mike Schwartz, Mike99999, MikeSy, Mikeblas, Mikey180791, MilerWhite, Millermk, Milo99, Mindmatrix, Minimac, Minna Sora no Shita, Mkeranat, Moazzam chand, Mojo Hand, Moreschi, Morwen, MrNoblet, Mrozlog, Mrt3366, Mspraveen, Mugaliens, Mukherjeeassociates, Mulad, Mumonkan, Mushroom, Mwaci11, Mxn, N1RK4UDSK714, N25696, NAHID, NSR, Nafclark, Namlemez, Nanshu, NathanBeach, NawlinWiki, NetManage, Netizen, NewEnglandYankee, Ngpd, Nick, Nicoosuna, Niteowlneils, Nk, Noah Salzman, Noctibus, Noldoaran, Northamerica1000, Northernhenge, Nsaa, Nurg, Ocaasi, Oda Mari, Odavy, Oddbodz, Ohka-, Oho1, Oli Filth, Olinga, OllieFury, OnePt618, OrgasGirl, Oroso, OverlordQ, PJM, PaePae, Pak21, PappaAvMin, Parzi, PatrikR, Paul August, Paul Drye, Paul E Ester, Paul Foxworthy, Paulinho28, Pcb21, Pdcook, Peashy, Pee Tern, PeeAeMKay, Pengo, PeregrineAY, PeruvianIlama, Pete1248, Peter Karlsen, Peter.C, Pgk, Phantomsteve, Pharaoh of the Wizards, Phearlez, PhilKnight, Philip Trueman, Philippe, Phinicle, Phoenix-wiki, Piano non troppo, Pillefj, Pingveno, Pinkadelica, Pjoef, Plrk, Pnm, Poeloq, Pol098, Poor Yorick, Poterxu, Praba tuty, Prabash.A, Prari, Prashanthns, Pratyya Ghosh, PrePress, Preet91119, Proofreader77, Prunesqualer, Psaajid, Psb777, Puchiko, Pvjohnson, Pyfan, Quadell, Qwertykris, Qwyrxian, Qxz, R'n'B, R3miixasim, RIH-V, RadioFan, RadioKirk, Rafaelschp, Railgun, Rakeki, Raspalchima, Ravinjit, Ray Lightyear, RayGates, RayMetz100, RazorXX8, Rdsmith4, Reaper Eternal, Reatlas, Refactored, Regancy42, Reidh21234, RenamedUser01302013, Rettetast, RexNL, Rhobite, Rich Farmbrough, Ricky81682, Ringbang, Rishu arora11, Riverraisin, Rj Haseeb, Rjwilmsi, Robert Merkel, Robert Skyhawk, Robocoder, Robth, Rocketrod1960, Rockonomics, Rohitj.iitk, Romanm, Rotanagol, Rothwellisretarded, Roux, Rursus, Ruud Koot, Ryager, Ryanslater, Ryanslater2, Ryulong, S.K., SAE1962, SDSWIKI, SFK2, SJP, SWAdair, Sae1962, Saiken79, Salvio giuliano, Sam Barsoom, Sam Korn, SamJohnston, Samir, Sander123, Sandman, Sango123, Sarchand, SarekOfVulcan, Satellizer, SatuSuro, Saturdayswiki, Savh, ScMeGr, Sceptre, Seanust 1, Seaphoto, SebastianHelm, Serketan, Several Pending, Sewebster, Shadowjams, Shadowseas, Sheeana, Shipmaster, Shirulashem, Siebren, Silly rabbit, Simeon, Simetrical, SimonMorgan, Sintaku, Sir Nicholas de Mimsy-Porpington, Sissi's bd, Siteobserver, Sjakkalle, Sjc, Skamecrazy123, Skybrian, Slakr, Sleske, SnoFox, Somchai1029, Sonett72, Sonia, Soosed, Sophus Bie, Soumark, SpK, Spartaz, Spazure, Spdegabrielle, SpikeTorontoRCP, SpuriousQ, Squids and Chips, Srdju001, Srikeit, Ssd, StaticVision, Stdazi, Stephen Gilbert, Stephenb, Stevertigo, Stifle, Stirling Newberry, Storm Rider, Strike Eagle, Strongsauce, Stuhacking, Sucker666, Sudarevic, Suffusion of Yellow, SuperHamster, Supertouch, Supreme Deliciousness, Supten, SwisterTwister, SymlynX, Sythy2, Tabletop, Tablizer, TakuyaMurata, TalkyLemon, Tasc, Tazmaniacs, Technopat, Tgeairn, Th1rt3en, Thatperson, The Anome, The Thing That Should Not Be, The Wiki Octopus, The wub, TheGrimReaper NS, TheNewPhobia, Thedjatclubrock, Thehulkmonster, Theimmaculatechemist, Theodolite, Theory of deadman, Thingg, Think outside the box, Thinktdub, Thomasryno, ThumbFinger, Thumperward, Tictacsir, Tide rolls, Tim Q. Wells, TimBentley, Title302, TittoAssini, Tobias Bergemann, Tolly4bolly, Tomatronster, Tonydent, Toquinha, Tpbradbury, Treekids, TrentonLipscomb, Trevor MacInnis, Triwbe, Troels Arvin, Trusilver, Tualha, Tudorol, Tuhl, Turlo Lomon, Turnstep, Twebby, Twelvethirteen, TwistOfCain, TwoTwoHello, Twsx, TyA, Tyler, UberScienceNerd, Ubiq, Ubiquity, Ugebgroup8, Ulric1313, Ultraexactzz, Uncle Dick, Unyoyega, VNeumann, Vary, Velella, Verajohne, Versus22, Veryprettyfish, Vespristiano, Victor falk, Vikreykja, Vincent Liu, Vipinhari, Vishnava, Visor, Vivacewwxu, VoxLuna, Vrenator, W mccall, W163, WOSlinker, Waggers, Waveguy, Wavelength, Weetoddid, Welsh, Werdna, Widefox, Widr, Wifione, Wik, Wiki alf, Wiki tiki tr, Wikidrone, Wikipelli, WikiuserNI, Willking1979, Wimt, Windsok, Winterst, Wipe, Wmahan, Woohookitty, Woseph, Writeread82, Wulfila, Wwmbes, Wya 7890, Xfact, Xhelllox, Xin0427, Yintan, Yossman007, ZenerV, Zhenqinli, Zipircik, Zippanova, ZooPro, Zro, Zundark, Zzuuzz, •, – — ˜, €•,ƒ „ …†‡, € • , , 3114 anonymous edits

Database model *Source*: http://en.wikipedia.org/w/index.php?oldid=570714755 *Contributors*: AGK, ARUNKUMAR P.R, AgadaUrbanit, Airplaneman, Alansohn, Amy whattt, AutumnSnow, Beland, Bill Slawski, CharlesBarouch, Cybercobra, Danim, Decrease789, Dkwebsub, Dwils098, ENeville, Edward, FatalError, J.delanoy, Jabbba, Jbolden1517, JoyMundy, Jwoodger, LaosLos, Magomaitin, MainFrame, Mark Renier, Materialscientist, Mdd, Mihai-gr, Mindmatrix, Minimac, Minna Sora no Shita, Mr. Vernon, Nn123645, Porterjoh, Razorbliss, Richramos, Roenbaeck, Sun Creator, Tim1357, Vegaswikian, Wikipelli, Woohookitty, 81 anonymous edits

Database normalization *Source*: http://en.wikipedia.org/w/index.php?oldid=577707258 *Contributors*: 1exec1, 4pq1injbok, A3 nm, ARPIT SRIVASTAV, Ahoerstemeier, Akamad, Akhristov, Alai, Alasdair, Alest, Alexey.kudinkin, Alpha 4615, Amr40, AndrewWTaylor, Antonielly, Anwar saadat, Apapadop, Arakunem, Arashium, Archer3, Arcturus, Arthena, Arthur Schnabel, Ascend, AstroWiki, AubreyEllenShomo, Autocracy, AutumnSnow, Azhar600-1, BMF81, Babbling.Brook, Bernard Fran™ois, Bewildebeast, Bgwhite, Billben74, Billpennock, BillyPreset, Black Eagle, Blade44, Blakewest, Blanchardb, Bloodshedder, Blowdart, BlueNovember, BlueWanderer, Bongwarrior, Boson, Bovineone, BradBeattie, Brick Thrower, BrokenSegue, Bruceshining, Bschmidt, Bugsbunny1611, BuzCo, CLW, Callavinash1, Can't sleep, clown will eat me, Chairboy, Chrislk02, Citral, CI22333, CodeNaked, Combatentropy, Conversion script, Creature, Crenner, Crosbiesmith, DARTH SIDIOUS 2, Damian Yerrick, DanMS, Dancraggs, Danim, Danlev, Datasmid, David Colbourn, DavidConrad, DavidHOzAu, Davidhorman, Dean001, Decrease789, Demosta, Denisarona, DerHexer, Dfass, Dflock, Discospinster, DistributorScientiae, Doc vogt, DocRuby, Docu, Don Hammond, Doud101, Dqmiller, Dreftymac, Drowne, Dthomsen8, Duke Ganote, Ed Poor, Edward Z. Yang, Eghanvat, Elcool83, Electricmuffin11, Elwikipedista, EmmetCaulfield, Emperorbma, Emw, Encognito, Enric Naval, Epepke, Eric Burnett, Escape Orbit, Ethan, Evilyuffie, Ewebxml, Falcon8765, Farquaadhnchmn, Fathergod, FauxFaux, Fieldday-sunday, Fireman biff, Flewellyn, Fluffernutter, Fmjohnson, Fraggle81, Fred Bradstadt, Furrykef, Gadfium, GateKeeper, Gimboid13, Ginsuloft, Gk5885, Gogo Dodo, Gottabekd, Gregbard, GregorB, Groganus, Gustavb, Guybrush, HMSSolent, Hadal, Hairy Dude, Hanifbbz, Hapsiainen, HbJ, Hbf, Heracles31, HiDrNick, Hoo man, Hu12, Hydrogen Iodide, Hz.tiang, Ianblanes, IceUnshattered, Imre Fabian, Inquam, Intgr, Jadvinia, Jakew, James086, JamesBWatson, Jamesday, Jamesjusty, Jan Hidders, Japo, Jarble, Jason Quinn, Javert16, Jdlambert, Jgro, Jjjjjjjjjj, Jklin, Joness59, Joseph Dwayne, Jpatokal, Jpo, Justin W Smith, KAtremer, KathrynLybarger, Keane2007, Keegan, KevinOwen, KeyStroke, Keyvez, Kgwikipedian, Kingpin13, Klausness, Kushalbiswas777, L Kensington, L'Aquatique, LOL, Larsinio, Lawrence Cohen, Leandrod, Lee J Haywood, Legless the oaf, Leleutd, Leotohill, Lerdthenerd, Les boys, Lethe, Libcub, Lifeweaver, Linhvn88, LittleOldMe, Longhair, Lssilva, Lujianxiong, Lulu of the Lotus-Eaters, Lumingz, Luna Santin, M4gnum0n, MER-C, Magantygk, Manavkataria, Mark Renier, Marknew, MarownIOM, MartinHarper, Masterstupid, Materialscientist, Matmota, Matthew 1130, Mckaysalisbury, Metaeducation, Michael Hardy, Michalis Famelis, Michealt, Microtony, Mike Rosoft, Mikeblas, Mikeo, Mindmatrix, Miss Madeline, Mjhorrell, Mo0, Modeha, Mooredc, Mpd, Mr Stephen, MrDarcy, MrOllie, Nabav, NawlinWiki, Nick1nildram, NickCT, NoahWolfe, Nocat50, Noisy, Northamerica1000, Nsaa, NubKnacker, Obradovic Goran, Ocrow, OliverMay, Olof nord, Opes, Oxymoron83, Pagh, Peachey88, Pearle, Perfectblue97, Pete142, Pharaoh of the Wizards, Phil Boswell, Philip Trueman, Pie Man 360, Pinethicket, Plastic rat, Polluxian, Prakicov, ProveIt, Purplepiano, Quarl, RB972,

RBarryYoung, RadioFan, Railgun, Rathgemz, Rdsmith4, Rdummarf, RealityApologist, Reedy, Regancy42, Reinyday, Remy B, Reofi, RichF, Rjwilmsi, Robert McClenon, Robomaeyhem, Rockcool19, Rodasmith, Romke, Ronfagin, Rp, Rumplefish, Ruud Koot, Ryulong, Sam Hocevar, Sasha.sheinberg, SchuminWeb, ScottJ, Scwlong, Seaphoto, Sfnhltb, Shadowjams, Shakinglord, Shawn wiki, Shreyasjoshis, Shyamal, Silpi, Simeon, Simetrical, Sixpence, Skritek, Smjg, Smurfix, Snezzy, Snigbrook, Socialservice, Sonett72, Soulpatch, Soumyasch, Spacesoon, Sstrader, Stacyshaelo, Stannered, Starwiz, Stephen e nelson, Stephenb, SteveHL, Stifle, Stolkin, Strike Eagle, Sue Rangell, Superjaws, Sydneyw, Sylvain Mielot, Szathmar, Taw, Tbhotch, Tcamacho, Tedickey, Teknic, Tgantos, Thane, The Thing That Should Not Be, The undertow, The1physicist, Tide rolls, Titofhr, Tobias Bergemann, Toddst1, Tom Lougheed, Tom Morris, Tommy2010, Toxicwaste288, Traxs7, Troels Arvin, Turnstep, Twinney12, Tyc20, Unforgettableid, Upholder, Utcursch, Vald, Valdor65, Vampyrium, VanishedUserABC, Velella, VinceBowdren, Vladsinger, Vodak, Voidxor, Waggers, WakimakiroIls, Wammes Waggel, Wavelength, Wexcan, WikiPuppies, WikipedianYknOK, Wildheat, Wilfordbrimley, Wilsondavidc, Winterst, Wjhonson, Woohookitty, WookieInHeat, Xiong Chiamiov, Xiroth, Yong-Yeol Ahn, Zedla, Zeyn1, Zhenqinli, Zzuuzz, ƒ „ …, 1332 anonymous edits

Database storage structures *[Source*: http://en.wikipedia.org/w/index.php?oldid=565777290 *[Contributors*: Abdull, Alai, Andrewman327, Beland, Decrease789, ElKevbo, Grafen, Jaytwist, Lenshapir, Mark Renier, Mskfisher, Rocketrod1960, Rursus, Troels Arvin, TubularWorld, 21 anonymous edits

Distributed database *[Source*: http://en.wikipedia.org/w/index.php?oldid=575281332 *[Contributors*: Alansohn, Ammubhave, Anthony, Arthur Rubin, Beland, Bomazi, Bporopat, CanisRufus, Centrx, Compfreak7, Danim, Derbeth, Dewritech, Donhalcon, Dpkade, Eastlaw, Eliz81, Gary King, Gensanders, GeorgeBills, Gregbard, Hooperbloob, Hu, Intelligentfool, Intgr, JCLately, Jamelan, Jandalhandler, Jason.yosinski, Jim1138, KeyStroke, Kku, Kuteni, Lguzenda, LilHelpa, M4gnum0n, Magioladitis, MelRobinson, Mere Mortal, Michaellacorte, Miym, Mschlindwein, Nikhil search, Nivix, Nonnompow, Owenja, Ozsu, Passport90, Pebkac, Perfecto, PeruvianIlama, PigFlu Oink, Prasanna8585, Ramaksoud2000, Satellizer, Sboehringer, Shibaji.paul, Sparky132, Squiddy, Sun Creator, Tempodivalse, Terry1944, TheThomas, Uncle Dick, Vektor330, Wbigger, Wizgha, 161 anonymous edits

Federated database system *[Source*: http://en.wikipedia.org/w/index.php?oldid=571954221 *[Contributors*: Beland, Bovineone, Cantonnier, Chris the speller, Comps, DBigXray, Dfoxvog, Frap, Gilo1969, Hu12, Joy, Khazar2, KingsleyIdehen, Kku, MacTed, Mark Renier, Martarius, Meena610, Mgh12, P. Dantressangle, Pmehra5730, Pumba lt, R'n'B, Repentsinner, Rettetast, Ringbang, Rjwilmsi, Sfan00 IMG, Shyamal, Tabletop, Tankiitr, The Thing That Should Not Be, TheParanoidOne, Threazy, Uthbrian, VanishedUserABC, Venullian, Woodshed, Yann Gripay, 63 anonymous edits

Referential integrity *[Source*: http://en.wikipedia.org/w/index.php?oldid=576386750 *[Contributors*: A3 nm, Aim Here, Allens, Amux, Andy Dingley, AnubisAscended, AutumnSnow, BL, Bearcat, Brandon, Brick Thrower, Daniel.Cardenas, Darkunor, Daviburg, DavidLevinson, Elwikipedista, Excirial, FatalError, Flon22, Friendlydata, Greentryst, I dream of horses, KeyStroke, Kmarshba, Lost tourist, Mark Renier, Materialscientist, Michael.Urban, Mindmatrix, Mtking, Muslim lo Juheu, Nburden, Neurolysis, Niceguyedc, Nivix, Obradovic Goran, Omicronpersei8, PatrickJCollins, Penartur, Philip Trueman, Psb777, Reatlas, Reedy, RuM, Sae1962, Sam Hocevar, Sietse Snel, Simtay, Snodnipper, Staszek Lem, Suvs2011, Ta bu shi da yu, Tarquin, Tolly4bolly, Varuna, Wjhonson, Wlievens, 113 anonymous edits

Relational algebra *[Source*: http://en.wikipedia.org/w/index.php?oldid=574653092 *[Contributors*: 2620:0:1002:1003:C06E:62E9:72C9:FDA2, Agquarx, Alain Amiouni, Alan Liefting, Alansohn, AlecTaylor, AndrewWarden, Anuj royal, Arthur Rubin, Arunloboforever, Austinflorida, AutumnSnow, Banazir, BiT, Blahedo, Blaisorblade, Brick Thrower, Bug, CALR, CRGreathouse, Cdrdata, Charvest, Chewings72, Chocolateboy, Chris the speller, Clawed, Cmdrjameson, Combatentropy, Cometstyles, CountMacula, Cryout, Cybercobra, Cycchina, DaveVoorhis, Davidfstr, Davnor, Derbeth, Dessources, Dhanuthilaka, DoriSmith, Download, Drowne, Drunken Pirate, EagleFan, Ed g2s, Edcolins, Egmontaz, Egriffin, Elektron, Elwikipedista, Esalder, Ezrakilty, Fabian Pijcke, Falcor84, Flyhighplato, Fresheneesz, FuFoFuEd, Gazpacho, Geira, Giftlite, Gregbard, GregorB, Hadal, Hans Adler, Hasanv, Hughitt1, Hussaibi, Hypergraph, IceCreamAntisocial, Infestor, IvanLanin, JackPotte, Jan Hidders, Jan1nad, Jarble, Javert16, JingguoYao, Jleedev, Joebolte, JohnyDog, Jon Awbrey, Joseph Dwayne, Jsnx, Juansempere, Justin W Smith, Kanenas, Keegan, KelvSYC, Khalid hassani, Kinaro, Kjetil r, Kku, Klausness, KnightRider, LOL, Lambiam, Larsinio, Leaflord, Lemycanh, Lfstevens, LtWorf, Magic5ball, Maksim-e, Mandries, Mani1, Mark Renier, Matthiaspaul, Mckaysalisbury, Mcthree, Mdd, Mets501, Michael Hardy, Michealt, Mikeblas, Mindmatrix, Msnicki, Myheimu, Nbarth, NewEnglandYankee, Ntmatter, O.Koslowski, Ocranom, Oleg Alexandrov, PanagosTheOther, PeruvianIlama, Peter.vanroose, Pgan002, Phamthelong, Polluxian, Popol1991, Qwertyus, R'n'B, Rathgemz, Reedy, Rgrimson, Rishig327, Rjwilmsi, Rleyton, Rsrikanth05, Ruakh, Rursus, Salix alba, Sam Staton, Samppi111, Sbrenesms, Scf1984, Schmid, Sdorrance, ShadowPhox, Shreyasjoshis, Sir Nicholas de Mimsy-Porpington, Slgcat, SnowFire, Sspecter, Stephan202, Tablizer, Tgeairn, The undertow, Tijfo098, Tommy2010, Tompsci, Troels Arvin, Twimoki, Vaucouleur, Vegpuff, Viperlight89, Wavelength, Way2veers, Wayne Slam, We64, Wifki, Wikipelli, Wrp103, Xcpenguin, Yoosofan, Zink Dawg, Ziusudra, † ‡ ˆ ‰ Š ‹, 343 anonymous edits

Relational calculus *[Source:* http://en.wikipedia.org/w/index.php?oldid=541136897 *[Contributors*: AutumnSnow, Cdrdata, Elwikipedista, Gregbard, Guppyfinsoup, Jan Hidders, Jim1138, Joieko, Jpbowen, Kku, Leandrod, Lfstevens, Mark Renier, Michael Hardy, Mikeblas, Mindmatrix, Omnipaedista, Opabinia regalis, Pewwer42, Remuel, Robert L Pendleton, Rsrikanth05, SqlPac, TheTito, 26 anonymous edits

Relational database *[Source*: http://en.wikipedia.org/w/index.php?oldid=577025448 *[Contributors*: *Kat*, 01001, 127, 217.162.105.xxx, 2620:0:1000:1402:6E3B:E5FF:FE0D:9B37, 64.192.12.xxx, Abdull, Abolen, Adamcscott, Adamrush, Admrboltz, Agateller, Ahoerstemeier, Alain Amiouni, Alansohn, Andre Engels, Angela, Anuja297, Appzter, Astheg, AutumnSnow, Avoided, Banes, Beland, Benderjaii, Beno1000, Bitnine, Bobo2000, Boothy443, Booyabazooka, Bpalitaa, Brennamack, Brick Thrower, Bsdlogical, CALR, Calmer Waters, Calvernaz, Chris.Giles, Chrislk02, Conversion script, Cpiral, Craig Stuntz, Crosbiesmith, Cww, DARTH SIDIOUS 2, DVdm, Dandv, Danim, Dannydaman9, Darth Mike, Dave6, David Delony, Dfeuer, DinosaursLoveExistence, Dionyziz, Dirk P Broer, DoorsAjar, Download, Drgs100, Dschwart11, Dumbledad, EagleFan, Eik Corell, El C, ElKevbo, Emperorbma, Fabrictramp(public), FatalError, FayssalF, Ferkelparade, Fidimayor, Fieldday-sunday, Filiocht, Findling67, FlyingDoctor, Francs2000, Fratrep, Fred Bradstadt, Freediving-beava, Frigotoni, Fuddle, Gaur1982, Gerbrant, Giftlite, Glane23, GoingBatty, Graham87, HJ Mitchell, Hapsiainen, Harold f, Herostratus, Hmrox, Hp-NJITWILL, I do not exist, ILikeBeer, IRP, Ideogram, Iohannes Animosus, J.delanoy, JCLately, JLaTondre, JaGa, Jacobrothstein, Jan Hidders, Jarble, Jitendraapi, Jncraton, John Vandenberg, Johnuniq, Jon Awbrey, Jwoodger, Jšna › šrunn, K faiad, KingsleyIdehen, Klausness, KnowledgeOfSelf, Kostmo, Kraron, Kristensson, Krogstadt, Kuru, Lalapicklem, Lamp90, Larsinio, Leandrod, Lfstevens, Linlasj, Logthis, Looxix, Luna Santin, LyricalCat, MC MasterChef, MER-C, Mac, MainlyDigGrammar, Mandarax, Manop, Mark Renier, Mark T, Mav, Mblumber, Mckaysalisbury, Merlion444, Metroking, Michael Hardy, Michael Hodgson, Mikeblas, MilerWhite, Mindmatrix, Msikma, NHRHS2010, Nannahara, Nanshu, Nisantha04, Niteowlneils, Nocohen, Ns.code, Odie5533, Odysseus1479, Olinga, Oursinees324, OwenBlacker, Oxymoron83, Pablo323, Pdcook, Pearle, Philcha, Pietdesomere, Pinkadelica, Psb777, Psychcf, Quitchy, Rasmus Faber, RayGates, Rchertzy, Rfl, Riggz0rer, Romanm, Rrburke, SandyGeorgia, Scouser0phil, Sequologist, Sfe1, Sgiovannini, Shinju, Sir Nicholas de Mimsy-Porpington, Sir Vicious, Sjschultz, Slightlyusefulcat, Smjg, Solipsist, Sonett72, Specialbrad, Spiritia, Spudtater, SqlPac, Stare at the sun, SteinbDJ, Steve Casburn, Stryn, Super48paul, Supten, TJRC, Tcncv, Tcnuk, Ted Longstaffe, Teles, TheDJ, Thingg, Tijfo098, Tobias Bergemann, Todd Vredevoogd, Tom harrison, Triddle, Triwbe, Troels Arvin, Ttguy, Turnstep, Utcursch, Vespristiano, Wavelength, Wesley, Wfrys, Wolfraem, Wolfsbane2k, Xiong, Xphile2868, Zahid Abdassabur, Zipircik, 580 anonymous edits

Relational database management system *[Source*: http://en.wikipedia.org/w/index.php?oldid=576713352 *[Contributors*: 16@r, AMD, Abb615, Acrider, Afabbro, Aldie, Ale And Quail, Altenmann, Anastrophe, Anvish, Anwar saadat, Apokrif, Athaenara, AutumnSnow, BL, BMF81, Ballin Insane10, Beland, Bgibbs2, Bob hoskins, BodyTag, Bonadea, Borgx, Brassrat70s, Bressan, Brick Thrower, Brilliantwiki, Cactus26, Chaitrabhat7, Chris Roy, Cnb, Craig Stuntz, Crosbiesmith, Cryptic, Cwitty, DB 103245, Darx9url, Davedx, Daverocks, Dewritech, DigitalEnthusiast, Dockurt2k, Elwikipedista, EoganOD, Faizan, FatalError, Fatehyab ahmed, Flata, George Rodney Maruri Game, Grunt, Gurch, HEAdrian, Heimstern, II MusLiM HyBRiD II, Igoldste, Igor Yalovecky, Ikhzter, J.delanoy, J36miles, JCLately, JFM, JHMM13, JamesBWatson, Jameshfisher, Jan Hidders, Jdthood, Jnlin, Joao.matos, Josemanimala, Joseph Dwayne, Joseph chennai, Jtgerman, Kaihsu, Karada, Kate, Kernel.package, KeyStroke, Kingston Dominik, Klausness, Kotika98, Kuru, Larsinio, Leandrod, Lfstevens, LinguistAtLarge, Lowellian, Lulu of the Lotus-Eaters, Mangoe, Mark Renier, Maximaximax, Mckaysalisbury, MelbourneStar, MikeSchinkel, Mikeblas, Mindmatrix, Minghong, Mintleaf, Mr4top, Mxn, Neilc, Nicks100, NuclearWarfare, Nylex, Oberiko, Obradovic Goran, Ohnoitsjamie, Ohyoko, Palica, Paulcolmer, Payal2820, PhiLiP, Pi, Pichpich, Pratyya Ghosh, Quest for Truth, RedHillian, Reddi, Reedy, Rfl, Rhobite, Robert Brockway, Rror, Sasquatch, Setppo, Shabbirbhimani, Shepazu, Smyth, Sparkiegeek, SqlPac, StevegiacomelIi, Szajd, Tablizer, TallMagic, TenPoundHammer, Tolly4bolly, Tomcat66 g500, Troels Arvin, Turnstep, UnDeRTaKeR, Uriber, Useight, VTPG, Vanished user qkqknjitkcse45u3, Vegaswikian, Vincent Liu, WIKIWIZWORKER, Wykypydya, Xcasejet, Xphile2868, œ•ž Ÿ ¡¢ž¢£, 382 anonymous edits

Relational model *[Source*: http://en.wikipedia.org/w/index.php?oldid=573775447 *[Contributors*: 130.94.122.xxx, 62.114.199.xxx, A930913, Adamcscott, Altenmann, AndrewWTaylor, AndrewWarden, AndyKali, AnonMoos, Arthur Rubin, Ashrust, Asukite, Audiodude, AutumnSnow, Aytharn, BD2412, BMF81, Babbling.Brook, Bblfish, Beland, Bento00, Bobo192, BonsaiViking, Brick Thrower, Brion VIBBER, Budloveall, CBM, Cadr, Cathy Linton, Cconnett, ChaosControl, Chessphoon, Chrisahn, Chrissi, Conti, Conversion script, Craig Stuntz, Crashoffer12345, Crosbiesmith, DARTH SIDIOUS 2, Damian Yerrick, Danim, DaveVoorhis, David Eppstein, Derek Ross, Dreadstar, Drunken Pirate, EagleFan, Ehn, Elwikipedista, Emx, Enric Naval, Erik Garrison, Evildeathmath, Furrykef, Fyrael, Gadfium, Gary D, Gary King, Giftlite, Gilliam, Grassnbread, Greenrd, Gregbard, GregorB, Gurch, Hans Adler, Helvetius, Hyacinth, Ideogram, Iluvitar, Immunize, Irishguy, Ixfd64, J04n, JCLately, Jadedcrypto, Jalesh, Jan Hidders, Jarble, JboIden1517, Jeff3000, JesseW, Jklowden, Jmabel, Joelm, Jon Awbrey, Jpbowen, Kassie, Kendrick Hang, Ketiltrout, Khalid hassani, Kimchi.sg, Kjkolb, Klausness, Korrawit, Lahiru k, Larsinio, Leandrod, Leifbk, Lethe, Lfstevens, Lopifalko, MER-C, Madjestic, Magioladitis, Maokart444, MarXidad, Marc Venot, Mark Renier, Materialscientist, Matt Deres, Mblumber, Mckaysalisbury, Mdd, Metaeducation, Mets501, Mhkay, Michael Hardy, MilerWhite, Mindmatrix, Moogwrench, Muntfish, NSash, Nad, Nascar1996, Neilc, Niteowlneils, NonDucor, Nsd, Ocaasi, Ocrow, Ozten, Pablothegreat85, Paul Foxworthy, Pitix, Pmsyyz, Pol098, PsychoAlienDog, Quazak Zouski, R'n'B, Razorbliss, Rbrwr, Reedy, Reyk, RonaldKunenborg, Ronhjones, Rp, Rursus, Ruud Koot, S.K., Sae1962, Sdorrance, Seraphim, SeventyThree, Sietse Snel, Simetrical, SimonP, Sonett72, Spartan-James, Spellcast, SpuriousQ, SqlPac, SteinbDJ, Stevertigo, THEN WHO WAS PHONE?, Tatrgel, Teknic, The-G-Unit-Boss, Tjic, Tobias Bergemann, Tohobbes, Tony1, Toreau, Troels Arvin, Tualha, Turnstep, Vikreykja, Welsh, Wgsimon, Windharp, Winhunter, Wjhonson, Woohookitty, Zklink, 303 anonymous edits

Object-relational database *[Source*: http://en.wikipedia.org/w/index.php?oldid=567656167 *[Contributors*: Acatyes, AndrewWTaylor, Beland, Bohunk, Bryan Derksen, Cbayly, CesarB, ChandraASGI, Chikako, Cybercobra, Danim, Diberri, Dmsar, DougBarry, Dze27, Enric Naval, FatalError, JLaTondre, Jamelan, Jay, Jerome Charles Potts, LedgendGamer, Mark Renier, Maury Markowitz, Mdd, Mindmatrix, Mrwojo, Mwtoews, Nmushov, Nthomas, Oxymoron83, Pedant17, Premil, Razorbliss, Rouenpucelle, Rp, Rursus, SeanTater, Sergsav, S…maphore, Tedickey, Theking2, Thomas Willerich, Tijfo098, Tom, Turnstep, Vald, 73 anonymous edits

Transaction processing *[Source*: http://en.wikipedia.org/w/index.php?oldid=574602965 *[Contributors*: 16@r, Abdull, Adolphus79, Agateller, Akulkis, Alkamins, Atlant, Avb, Awolski, BBCWatcher, BD2412, Baiji, Beland, Beve, Bnicolae, Bruvajc, CaroleHenson, Cbwash, Chairman S., Charleyrich, Clausen, Cliffb, Craig Stuntz, CutOffTies, DGG, Danielle009, Danim, Donsez, Download, Ellynwinters, Gf uip, Ghaskins, Gordonjcp, GregRobson, Gutza, JCLately, JHunterJ, Jan1nad, Jmcw37, Jorgenev, Joshua Scott, Kgf0, Khalid hassani, Kubanczyk, Lear's Fool, Luıs Felipe Braga, M4gnum0n, MER-C, MONGO, Mandarax, Mark Renier, Maury Markowitz, Mika au, Mikeblas, Mindmatrix, MrOllie, Oo7nets, Oxymoron83, Pcap, Peter Flass, Pratyeka, Radagast83, Rbpasker, Rettetast, Ruud Koot, SEWilco, Stephan Leeds, Stymiee, Suruena, Tobias Bergemann, Tschristoppe, Unimath, Uzume, Wireless friend, Wtmitchell, Zippy, Zzuuzz, 106 anonymous edits

ACID *[Source*: http://en.wikipedia.org/w/index.php?oldid=577826570 *[Contributors*: 123Hedgehog456, 2001:18E8:3:11BC:20D:56FF:FE85:C51, Accurizer, Acdx, Af648, Agnt9, AlanUS, Amolshah, Andrei S, Anthony Appleyard, AxelBoldt, Barefootguru, Barrylb, Beland, Benandorsqueaks, Bernhard Bauer, Bezenek, BlueNovember, Bluiee, Boing! said Zebedee, BonsaiViking, Bunyk, CPColin, Ceyockey, Christian75, Clausen, Cole2, CorbinSimpson, DAllardyce, DHN, Daniel11, Dave.excira, Decibel, DevonDBA, Download, DragonLord, Drake Redcrest, Duncan.Hull, E2eamon, Edward, Elwikipedista, Endersdouble, Epbr123, Espoo, Excirial, FatalError, FayssalF, Flewis, Forderud, Fragglet, Fubar Obfusco, Fylbecatulous, Gakusha, Gf uip, Ghostdood, Gioto, Golfguy399, Gorgan almighty, GregRobson, Gurchzilla, HJ Mitchell, Haakon, Harsh 2580, Heiser, Hellknowz, Hmrox, I dream of horses, IMSoP, Inimino, Inter16, Intgr, It Is Me Here, Ivan Pozdeev, J.delanoy, JCLately, Jagun, Jaydlewis, Jeff G., Jessemerriman, Jim1138, Jleedev, Jontomkittredge, Joonasl, Jordonbyers, Jpbowen, Kainaw, Kam Solusar, Karada, Kd24911, Kirilldoom16, Kku, Kmorozov, Kristof vt, Larsinio, Lee Carre, Lfstevens, Loren.wilton, LuoShengli, Luıs Felipe Braga, MacMog, Maimai009, Makecat, Marek69, Mark Arsten, Mark Renier, Markonen, Martin451, Materialscientist, Matusz, Maury Markowitz, Mcherm, Mdann52, Michael Hardy, Mindmatrix, Miym, MrRedwood, Mrwojo, Mskfisher, Mynameissskr, Neilc, NewAspen1, Ngpd, Nmfon, Noformation, Noommos, Passargea, Paul Foxworthy, Paul Magnussen, Personman, Petiatil, PierreAbbat, Pip2andahalf, Polupolu890, Ponder, Pontificalibus, Poor Yorick, Prachee.j, Prasannawikis, Premsurya, Puffin, Quuxplusone, R'n'B, RUL3R, Raja99, Raztus, RedWolf, Redrose64, ReelIrt, Renku, Rfl, Rich Farmbrough, Rjwilmsi, Rlaager, Rob7139, Sae1962, Saeed Jahed, Safalra, Salix alba, Saucepan, Sean D Martin, SeanAhern, Seaphoto, Sesshomaru, Shenme, Shmuelsamuele, Siggimund, Siskus, Some jerk on the Internet, SpaceFlight89, SqlPac, Stangaa, StephanCom, Strike Eagle, Surturz, Suruena, Svick, Swamp Ig, Synchronism, Tabledhote, Tagus, Tct13, Tgeairn, Thecodysite1, ThinkerFeeler, Thomas Willerich, Throwaway85, Thumperward, Tide rolls, Titodutta, Tolly4bolly, Tommy2010, Trefork, Triesault, Trusilver, Trvth, Turnstep, UFu, Uiteoi, Urhixidur, Verloren, Vertium, Vhabacoreilc, Victor falk, Vina, Viridae, Vrenator, WadeSimMiser, WhiteOak2006, Wikipelli, Wildrain21, Winston Chuen-Shih Yang, Woo333, Wykypydya, Yazan kokash23, Yourbane, Ytcracker, Yurik, ZappaOMati, Zhenqinli, Zigger, Zippy, Zoicon5, •, 579 anonymous edits

Create, read, update and delete *[Source*: http://en.wikipedia.org/w/index.php?oldid=574555474 *[Contributors*: 5994995, Absinf, Alex2222, Alvin-cs, Ant, Antonielly, Apokrif, Banzaimonkey, Beatupbutterfly, Bovski, Chris Purcell, DaGizza, Demonkoryu, Dhartung, Donperk, Dreftymac, Eraldito, FeRD NYC, Fgrinder, Fluffernutter, Fred Bradstadt, Gary King, Garylhewitt, Gjs238, Gobbleswoggler, Gogo Dodo, Guoxiao281, Herbee, Iancarter, Intgr, James Harvard, Jarble, Jim1138, Jleedev, Jmkim dot com, Kazvorpal, Kbrose, KeithTyler, Kenyon, Khalid hassani, KnightRider, Korg, Kozuch, L Kensington, Leonus, Lord Zoner, LucQ, Luıs Felipe Braga, Lyverbe, Mark McColl, Mark Renier, MarkusStolze, Martnym, Max Terry, Metal.lunchbox, Mike Blackney, Mikeblas, Mindmatrix, Mr e guest79, Mzajac, Napoleonsacrebleu, NawlinWiki, Nepenthes, OguzOzkeroglu, Pinethicket, Pmcm, RJFJR, Railwayfan2005, Rayngwf, Remuel, Richnice, RickScott, Robert K S, RobertG, Robertbowerman, Sannse, Sawall, Stephan Leeds, Syhon, Techi2ee, Thorwald, Th'ringer, Troels Arvin, Unixxx, Vectro, Wesley, Winterst, Zanerock, Zegoma beach, 190 anonymous edits

Null (SQL) *[Source*: http://en.wikipedia.org/w/index.php?oldid=577846588 *[Contributors*: Abdull, Alejos, Andreas Kaufmann, Andylkl, Andyyso, Arcann, Beno1000, Bgwhite, Bradeos Graphon, Cedar101, Cybercobra, Daniel.Cardenas, Dudegalea, EJSawyer, Ehdrive11, Elwikipedista, Furrykef, Gaius Cornelius, Gregbard, GregorB, Halo, Harryboyles, Haus, HaywardRoy, Hu12, Igor Yalovecky, Incnis Mrsi, Iqbalhosan, Jdlambert, John of Reading, Jonathan de Boyne Pollard, Julesd, Koavf, Kobrabones, Langpavel, Lightmouse, LilHelpa, Loadmaster, Luıs Felipe Braga, MER-C, Mahahahaneapneap, Malleus Fatuorum, Mark Renier, Matozqui, Mblumber, MeekMark, Michael Hardy, Mikeblas, Mindmatrix, Modify, Mwtoews, Myheimu, Nigelj, Nirion, Northernhenge, Ntounsi, Ott2, Plustgarten, Quadell, Random832, Rich Farmbrough, Rjwilmsi, RockMFR, Ruakh, Senpai71, Simetrical, SmallRepair, Smith609, SnappingTurtle, SqlPac, Stolze, Tasdevil13, Terrificfriffid, Teukros, The Fortunate Unhappy, Three-quarter-ten, Tijfo098, Tony1, Visor, Voer, Xoneca, Zeeyanwiki, Zhenqinli, 109 anonymous edits

Candidate key *[Source*: http://en.wikipedia.org/w/index.php?oldid=575890156 *[Contributors*: Acroterion, AlexPlank, Amikake3, AndrewWarden, Arravikumar, Axiomsofchoice, Brick Thrower, Bryant1410, Captmjc, Charles Matthews, Crosbiesmith, CyborgTosser, DMacks, DVdm, Dharmabum420, Docu, Ejrrjs, Eric22, FuthaMukker, Hans Adler, HenningThielemann, J.delanoy, Jan Hidders, Jleedev, Jorge Stolfi, Josephbui, JoshDuffMan, Kalyson, KeyStroke, LanguageMan, Mark Renier, Massic80, Materialscientist, MiloszD, Mindmatrix, Mwtoews, Nabav, Neilc, Obradovic Goran, Patriotic dissent, Possession, ProcerusDecor, Prodizy, Rbrewer42, Rholton, Richaraj, Ronald S. Davis, SqlPac, Sreyan, Stolkin, Torzsmokus, Weedwhacker128, Yahya Abdal-Aziz, ZeroOne, •, ƒ „ …, 96 anonymous edits

Foreign key *[Source*: http://en.wikipedia.org/w/index.php?oldid=577704950 *[Contributors*: 16@r, Abdull, Amix.pal, Anbu121, AndrewWarden, Arichnad, Arthur Schnabel, Arunsinghil, Aurochs, AutumnSnow, Beesforan, Biochemza, Brick Thrower, Can't sleep, clown will eat me, Cander0000, Causa sui, Clarificationgiven, Cory Donnelly, Cpiral, Cww, DHN, Darth Panda, Derek Balsam, DireWolf, Dobi, Dougher, EWikist, Eldavan, Electricnet, Entropy, FatalError, Feder raz, Fluffernutter, Flyer22, Frap, Govorun, GregorB, Gsm1011, IanHarvey, JHunterJ, Jadriman, Jesselong, Jim1138, Jk2q3jrklse, Jlenthe, Joebeone, John of Reading, KeyStroke, Kf4bdy, Kubntk, Larsinio, Marcusfriedman, Mark Renier, MexicanMan24, Mike Rosoft, Mikeblas, MikeyTheK, Mindmatrix, Minimac, Mmtrebuchet, Mogism, Mormegil, MrOllie, Mrt3366, Natalie Erin, Ngriffeth, O.Koslowski, Obradovic Goran, PPOST, Pbwest, Peak, Polypus74, RIL-sv, Reedy, Rjwilmsi, Rror, Rsrikanth05, SDS, Salvatore Ingala, Sboosali, Selfworm, Semperf, Shreyasjoshis, Species8473, Staecker, Stolze, Svenmathijssen, Tarquin, Tgeairn, The Thing That Should Not Be, TheExtruder, Threeacres, Timhowardriley, TobiasPersson, Troels Arvin, Unordained, Waskyo, WikHead, Zipz0p, •, ƒ „ …, 220 anonymous edits

Unique key *[Source*: http://en.wikipedia.org/w/index.php?oldid=571010506 *[Contributors*: Aberdeen01, Ahoerstemeier, Akyadav324, Alessandro57, Alexjbest, AmandeepJ, Ambuj.Saxena, Andre.psantos, Baa, Blue Square Thing, Boson, Causa sui, ChrisGualtieri, Ctimko, DarkFalls, DeadEyeArrow, Dgc03052, Dougher, Drphilharmonic, Ewebxml, Faizan, Feraudyh, Fnielsen, Frap, Gurch, Hike395, J.delanoy, JHunterJ, Jberkus, Jbodilytm, Jdeperi, Jesdisciple, Joe.dolivo, Jorge Stolfi, Jyothisdavid4u, KeithB, L'Aquatique, L337 kybldmstr, LittleOldMe, Loren.wilton, Mahemoff, Materialscientist, Mindmatrix, Minimac, Mwtoews, Nabav, Natalie Erin, Northamerica1000, O.Koslowski, Obradovic Goran, Pevernagie, Praveentech, ProcerusDecor, Raja200682, Ratarsed, RobIII, Spartaz, Special Cases, Spinality, Stolze, Subversive.sound, Themusicgod1, Thumperward, Tijfo098, TommyG, Troels Arvin, Unixxx, Velavan, Vjosullivan, Wensceslao, Whitejay251, Wiki.Tango.Foxtrot, Winterst, X201, Yintan, Ykliu, Zzuuzz, 113 anonymous edits

Superkey *[Source*: http://en.wikipedia.org/w/index.php?oldid=573338089 *[Contributors*: 2001:700:303:D:8DF3:FDC7:B975:9C41, 2001:700:303:D:BCEB:D496:EA00:FD55, AndrewWarden, Anog, AutumnSnow, Boson, CeleronNutcase, CharlotteWebb, ColinFine, Crosbiesmith, Dawynn, Fatherlinux, Fimp, Igor Yalovecky, IronGargoyle, James Crippen, Jan Hidders, Jorge Stolfi, Jusdafax, Jwulff, Katieh5584, KeyStroke, Kranix, LOL, Larsinio, M. Frederick, Magioladitis, Mark Renier, Metron4, Michaelcomella, Mikeblas, Millermk, Mindmatrix, Nabav, Pimlottc, ProcerusDecor, Reedy, Rhoerbe, SpuriousQ, Sss41, Stbrob, The Thing That Should Not Be, TheParanoidOne, Tobias Bergemann, Torzsmokus, Twarther, Voidxor, Welsh, Wikitanvir, Yay unto the Chicken, Zzuuzz, •x, •, ƒ „ …, 74 anonymous edits

Surrogate key *[Source*: http://en.wikipedia.org/w/index.php?oldid=572624859 *[Contributors*: 2001:4898:98:2041:2468:9C5:9E15:D5AF, Barliner, Brick Thrower, Bryant1410, ChrisNoe, Chrisxue815, DVdm, Darinw, Demitsu, Djankowski, Dtuinhof, Egrabczewski, Favonian, Govorun, Groggy Dice, Hairy Dude, Hsauzier, Int19h, Jberkus, Jimgawn, Joeharris76, KeyStroke, Kgaughan, Kjkolb, LachlanA, Leandrod, Lucianosother, M4gnum0n, Mark Renier, Mcbridematt, Mcclarke, Mdchachi, Mdd, Mindmatrix, MyTigers, Neilc, Pearle, PhiLiP, Phil Boswell, Pinkadelica, Raggatt2000, Reddyfire, Reedy, Rich Farmbrough, Rjwilmsi, Robert K S, Shadowjams, Shenme, Simetrical, Sleske, Stewartadcock, Template namespace initialisation script, Tfitzg, Tim.spears, Timhowardriley, Toh, Tomas e, Troels Arvin, Vjosullivan, WikipedianMarlith, Xenan, 160 anonymous edits

Armstrong's axioms *[Source*: http://en.wikipedia.org/w/index.php?oldid=575689045 *[Contributors*: A3 nm, Aednichols, Andonic, Arosa, CBM, Can't sleep, clown will eat me, Charles Matthews, ChrisGualtieri, Cornellcloud, Entropeter, Inklein, Jh559, Jonemerson, Joseph Dwayne, Loul, Mark Renier, Mentoz86, Paolo Serafino, Q-lio, Telofy, Tijfo098, Vegpuff, Wavelength, 62 anonymous edits

Relation (database) *[Source*: http://en.wikipedia.org/w/index.php?oldid=576215376 *[Contributors*: AndrewWarden, Asfreeas, Asocall, AutumnSnow, Crosbiesmith, Ed Poor, Fratrep, Georgeryp, Icairns, Lfstevens, MaD70, Mark Renier, MusiKk, NickCT, Nigwil, Rob Bednark, Subversive.sound, Tijfo098, Universalss, 14 anonymous edits

Table (database) *[Source*: http://en.wikipedia.org/w/index.php?oldid=571781837 *[Contributors*: 12george1, 16@r, Abdull, Ajraddatz, Alai, Arcann, AutumnSnow, Blanchardb, Bobo192, Bongwarrior, Bruxism, C.Fred, Cbrunschen, Correogsk, Cyfal, DARTH SIDIOUS 2, Danim, Dreftymac, Dzlinker, Epbr123, FattyMcjimmy, Feder raz, Funnyfarmofdoom, Gurch, IMSoP, IanCarter, J36miles, Jamelan, Jerome Charles Potts, Krishna Vinesh, Larsinio, LeonardoGregianin, Lfstevens, Mark Renier, Materialscientist, Mblumber, Mikeblas, Mikeo, Mindmatrix, Morad86, N0nr3s, Nibs208, Nikuwap, Ofus, Pyfan, Quentar, S.K., Sae1962, Scs, Senator2029, Sietse Snel, SimonP, Sippsin, Sonett72, SqlPac, Stolze, TheParanoidOne, TommyG, Turnstep, Txomin, Versageek, Widefox, Yug1rt, Zhenqinli, 101 anonymous edits

Column (database) *[Source*: http://en.wikipedia.org/w/index.php?oldid=575439689 *[Contributors*: AbsoluteFlatness, Arcann, CesarB, CommonsDelinker, Danim, Dreftymac, Frietjes, F", GermanX, Huiren92, Jmabel, KeyStroke, Mark Renier, Mark T, Mzuther, Petrb, RJFJR, Sae1962, Sietse Snel, SqlPac, ƒ „ …, 13 anonymous edits

Row (database) *[Source*: http://en.wikipedia.org/w/index.php?oldid=555504177 *[Contributors*: 2help, Allen3, Asfreeas, CommonsDelinker, D4g0thur, Danim, David H Braun (1964), Flip, GLaDOS, Gail, GermanX, Glacialfox, GregorySmith, Jamespurs, Jerroleth, Jmabel, KKramer, KeyStroke, Liujiang, Mark Renier, Mark T, Mxg75, Mzuther, O.Koslowski, Oyauguru, Pnm, Pol098, Retodon8, Rjd0060, Ronhjones, Shaka one, Sietse Snel, SootySwift, Troels Arvin, Yamamoto Ichiro, 29 anonymous edits

View (SQL) *[Source*: http://en.wikipedia.org/w/index.php?oldid=574141903 *[Contributors*: Abdull, Acjelen, Alai, Andrew.george.hammond, Anthony Appleyard, Blowdart, Boson, BullRunner2009, Cedar101, Christian75, ClementSeveillac, Dcoetzee, Dfrg.msc, Edwardzhu, Elcasc, Ewebxml, FernandoAires, Fragment, Galador, JDHeinzmann, JForget, JLaTondre,

JeepdaySock, Jerome Charles Potts, Joaquin008, Jobbin, Jtgerman, Jwoodger, Kibbled bits, Kku, Kuru, Larsinio, Mark Renier, Mathmo, Matinict, Mikeblas, Mindmatrix, MrOllie, Muchium, Nothings, Quentar, Quuxplusone, RaBa, Raeky, Rfl, Ricardo rivaldo, Rjwilmsi, Roux, Rsrikanth05, Ruud Koot, S.K., Sappy, Scrool, Sippsin, Smalljim, Stolze, TheDamian, TommyG, Toyota prius 2, Troels Arvin, UncleDouggie, WmLGann, Woohookitty, Wwphx, Z.E.R.O., Zerodeux, Zhenqinli, 144 anonymous edits

Database transaction *[Source*: http://en.wikipedia.org/w/index.php?oldid=576565184 *[Contributors*: 16@r, Adi92, Ajk, Al3ksk, AmandeepJ, AnmaFinotera, Appypani, Babbage, Binksternet, Burschik, CharlotteWebb, Clausen, Comps, Craig Stuntz, DCEdwards1966, Damian Yerrick, Daniel0524, Dauerad, Derbeth, DnetSvg, Forderud, Fratrep, Geniac, Georgeryp, Gerd-HH, Gf uip, Ghettoblaster, GregRobson, Haham hanuka, Hbent, Hede2000, Highguard, HumphreyW, Intgr, JCLately, Jarble, Jason Quinn, Jeltz, Karel Anthonissen, KellyCoinGuy, KeyStroke, Khukri, Larsinio, Leeborkman, Lingliu07, Lubos, Luc4, Lysy, M4gnum0n, Mark Renier, Matiash, MegaHasher, Mike Schwartz, Mikeblas, Mindmatrix, Mintleaf, Neilc, Nixdorf, OMouse, Obradovic Goran, Owen, Paul Foxworthy, Pcap, Pepper, RedWolf, RichMorin, Rocketrod1960, Roesser, SAE1962, Sandrarossi, SebastianHelm, Sobia akhtar, SqlPac, Stevag, T0m, Timo, Triwger, Troels Arvin, Turnstep, Wei¥Nix, Zerksis, Zhenqinli, 107 anonymous edits

Transaction log *[Source*: http://en.wikipedia.org/w/index.php?oldid=575149952 *[Contributors*: Clausen, DGG, Damian Yerrick, Gustronico, Intgr, JCLately, JLaTondre, KeyStroke, Larsinio, Lupin, Mark Renier, Mikeblas, Mindmatrix, Neoconfederate, Pelister, Poor Yorick, SJP, Sleske, SoledadKabocha, Stolze, Twimoki, 38 anonymous edits

Database trigger *[Source*: http://en.wikipedia.org/w/index.php?oldid=576465806 *[Contributors*: Abdull, Acha11, Adarshramesh, Bevo, BobHindy, Brick Thrower, Bucketsofg, Cadillac, Can't sleep, clown will eat me, Cedar101, ClamDip, ClanCC, CodeNaked, DanBishop, DanielcWiki, Deineka, Denisarona, Derbeth, Dffgd, Dirkbb, Fizalhaji, F", Grondemar, Gurch, HMSSolent, Hazard-SJ, Heron, Hu12, Jerome Charles Potts, John of Reading, Jyujin, Knakts, L337 kybldmstr, Larsinio, Lugia2453, M2Ys4U, Magioladitis, Mark Renier, Matinict, Mecanismo, Mike Rosoft, Mikeblas, Mindmatrix, Minna Sora no Shita, MIpearc, Mortense, MrOllie, Mschlindwein, NathanBeach, Nickleus, Niteowlneils, Noah Salzman, Noelweichbrodt, PaD, Pimlottc, Pinethicket, RJFJR, Ramkrish, Reedy, Rimonon, Ross Fraser, Rsrikanth05, S.K., Sae1962, Sampsonvideos, Sippsin, SluggoOne, Stolze, SuperHamster, Superhilac, Svick, Tlaresch, Troels Arvin, Unordained, Windofkeltia, Yourbane, Yrithinnd, ˜‰ˆ ˜ˆf, 295 anonymous edits

Database index *[Source*: http://en.wikipedia.org/w/index.php?oldid=577312945 *[Contributors*: 16@r, 31stCenturyMatt, Abolen, Afriza, Antandrus, Apavlo, Arcann, Arleyl, Arny, Atree, Aurlee, Bezenek, Brian Tvedt, Cander0000, Carmichael, Ccare, Ceyockey, Chamoquemas, Chire, ChrisGualtieri, CloudNine, ColinFrayn, Comps, Cybercobra, DJPohly, Dainis, Danlev, Deon Steyn, Dionyziz, Dominiktesla, Dougher, Drewnoakes, Dvik, Echawkes, Ercanyuz, ErikHaugen, Euryalus, Excirial, Flewis, Flyer22, Flyrev, Focus22, Furrykef, Gaur1982, Gergie, Glacialfox, Gnaaye, GordonFindlay, Groffg, Groves.w, Gwyant, InShaneee, Interiot, Intgr, JCLately, Jadecristal, Jamesjiao, Jasimab, Jerryji1976, Jfroelich, Jim CarnicelIi, Jivadent, Jlehew, JohnF1980, Jon Awbrey, Jschnur, Jspashett, Jwchong, Kayau, KnightRider, Kuru, Larsinio, Leuko, Lfstevens, Lsschwar, Mabuali, Machadoman, MahSim, Manishkarwa, Mark Renier, MarkusWinand, Mereman, Mets501, Mike Rosoft, Mindmatrix, Morfeuz, Movses, MrOllie, Mxcatania, M'slimix, NGPriest, Nahoo, NellieBly, Nepenthes, NicDumZ, Nicolas1981, Norm mit, Oxymoron83, P21n7, Pawanjain19, Pietrow, Planetneutral, Ppntori, Radagast83, Raypereda, Rich Farmbrough, RI, RobSimpson, Ruzihm, S.K., Salvio giuliano, Samroar, Samson ayalew, Sandgem Addict, Sbisolo, Searcherfinder, Sideswipe091976, SimonP, Sippsin, Sleske, SpaniardGR, SpeedyGonsales, Stefan Udrea, Stegop, THEN WHO WAS PHONE?, Taka, The Thing That Should Not Be, Tide rolls, Tommy2010, Triddle, Turnstep, TutterMouse, Vacio, Wbm1058, Wikiwikithe3rd, William Avery, Woohookitty, X7q, Yalckram, YamaguchiŒ • , Zhenqinli, 469 anonymous edits

Stored procedure *[Source*: http://en.wikipedia.org/w/index.php?oldid=574743100 *[Contributors*: 0goodiegoodie0, 3Jane, Aaadamaa, Abdull, Aleenf1, Amaury, Andreas Kaufmann, Andy.ruddock, Aravind V R, AvicAWB, Av…, Berny68, Bevo, BobHindy, Bobo192, Bovineone, Brenan99, Brianray, Calane83, Cedear, ChristopherGautier, ClementSeveillac, Coachbudka, Cww, DVdm, Dcoetzee, Derbeth, Dogsgomoo, Dougher, Drake Redcrest, Dreamofthedolphin, Duster.Cleaner, EdgeOfEpsilon, Elockid, EvanCarroll, EvanSeeds, Farazbs20, Favonian, Flashspot, Frap, Frecklefoot, Fred Bradstadt, Friendlydata, Gambhava, GoldenTorc, Graham87, GregorB, Harryboyles, Homestarmy, Honeplus, Hu12, IO Device, Ichimonji10, Izogi, JCLately, Jay, Jeffreyarcand, Jeltz, Jim1138, Jogloran, KeyStroke, Kmsimon, Kuru, Kvdveer, Kyledmorgan, Larsinio, Lewissall1, Lights, Luckypayal, M4gnum0n, MER-C, Mariolina, Markblue, Marr75, Martincamino, Materialscientist, Matticus78, Mayur, Merbabu, Michael@nosivad.com, Mikeblas, Mikesheffler, MilerWhite, Mindmatrix, Mitchandsherri, Modster, Moe Epsilon, MrJones, MrOllie, Mschlindwein, NYCDA, Neilc, Nickdc, Nsaa, Ohiostandard, Pedro, Petersap, Pinethicket, Pravs, Primalmoon, Pseudonym, Rajeearul, Red Thrush, Regani, RevRagnarok, Rich Farmbrough, Riki, Rror, Rythie, S. , rvarr.S, Sava chankov, Scadavidson, Scgtrp, Sdorrance, SimonP, Sippsin, Sqlinfo, Stevietheman, Stolze, SymlynX, Taeshadow, Thumperward, Tijfo098, Tobias Bergemann, Troels Arvin, Unyoyega, Velella, Winston Chuen-Shih Yang, Xenium, Xzilla, Zhenqinli, •, 440 anonymous edits

Cursor (databases) *[Source*: http://en.wikipedia.org/w/index.php?oldid=577352465 *[Contributors*: Abdull, Abi79, Agujero Negro, Aitias, Alik Kirillovich, BlastOButter42, Catgut, Cedar101, Christian75, Cwolfsheep, Danielx, DarkFalls, Darth Panda, DigitalEnthusiast, Dmccreary, Doug Bell, Ejdzej, Epbr123, Feder raz, Ffu, Fieldday-sunday, Greenrd, Habitmelon, Haleyga, Hutcher, Ilyanep, Ivantalk, Jamestochter, Janigabor, Jerome Charles Potts, Justinc, Kamots, Kubieziel, Larsinio, Mark Renier, Mikeblas, Mindmatrix, Miten.morakhia, MrOllie, Mwtoews, Nagae, NawlinWiki, NeonMerlin, NewEnglandYankee, OsamaK, PhilHibbs, Primalmoon, RHaworth, RandyFischer, Reedy, Richi, S.K., S. , rvarr.S, Sachzn, SarekOfVulcan, SimonP, Sippsin, SkyWalker, Stolze, Tbannist, TechTony, Tobias Bergemann, Underpants, Wallie, Winston Chuen-Shih Yang, Wjasongilmore, Wknight94, Xiphoris, 92 anonymous edits

Partition (database) *[Source*: http://en.wikipedia.org/w/index.php?oldid=577826075 *[Contributors*: Alai, Andrew.rose, Andy Dingley, Angusmca, Beaddy1238, Brian1975, Ccubedd, Ceva, Doubleplusjeff, Drrngrvy, Ehn, Fholahan, Foonly, Geoffmcgrath, Georgewilliamherbert, Habitmelon, Highflyerjl, Isheden, Jamelan, Jan.hasller, Jonstephens, Lurkfest, Mark Renier, Materialscientist, Mdfst13, Mikeblas, Mindmatrix, MrOllie, Peak, Pinkadelica, S.K., Salobaas, Semmerich, SmallRepair, Stevelihn, Vishnava, Wordstext, Yahya Abdal-Aziz, 42 anonymous edits

Concurrency control *[Source*: http://en.wikipedia.org/w/index.php?oldid=550119263 *[Contributors*: 2GooD, Acdx, Adrianmunyua, Augsod, Bdesham, Brick Thrower, CanisRufus, CarlHewitt, Christian75, Clausen, Comps, Craig Stuntz, Cyberpower678, DavidCary, Furrykef, Gdimitr, GeraldH, JCLately, Jesse Viviano, Jirislaby, John of Reading, JonHarder, Jose Icaza, Karada, KeyStroke, Kku, Leibniz, M4gnum0n, Magioladitis, Malbrain, Mark Renier, Mgarcia, Mindmatrix, Miym, N3rV3, Nealcardwell, Nguyen Thanh Quang, Peak, Poor Yorick, Reedy, Rholton, Ruud Koot, Siskus, Smallman12q, The Anome, Thingg, Thoreaulylazy, Tikuko, TonyW, Touko vk, Tumble, Victor falk, Vincnet, Wbm1058, Wikidrone, YUL89YYZ, 87 anonymous edits

Data dictionary *[Source*: http://en.wikipedia.org/w/index.php?oldid=564190594 *[Contributors*: Aednichols, Ajdlinux, AlistairMcMillan, Aluion, Andreas Kaufmann, Awis, Barek, BozMo, Bubba hotep, ChrisGualtieri, Cybercobra, DEddy, Dan100, Daniel5127, Daswani.Amit, Davnor, DePiep, Dekisugi, Dicklyon, Dmccreary, Floweracre, Friendlydata, Gilliam, Gioto, Giso6150, Gobbleswoggler, Hadal, Ham Pastrami, Hardyplants, Haymaker, Helwr, Hooperbloob, Icey, Immunize, Iridescent, Jacobko, Jaksckajwsb, Jamelan, Jeff3000, JeffTan, Joelemaltais, Joinarnold, Jwissick, Karada, Ketiltrout, KeyStroke, Kku, Klaun, Lauri.pirttiaho, M.r santosh kumar., Mark Renier, MarkDWikiUser, Materialscientist, MaxHund, Maziotis, Mdd, Mentifisto, Michael Hardy, Mindmatrix, Mushroom, N1RK4UDSK714, Olaf Davis, Omicronpersei8, PartyDude!, Pavel Vozenilek, Perrydegner, Pnm, RattusMaximus, RayGates, Rettetast, Riana, Rich Farmbrough, RickBeton, Shadowjams, Sprachpfleger, Sstrader, The Thing That Should Not Be, TheParanoidOne, Thetorpedodog, Tigerente, Ttwaring, Veyklevar, Violetriga, Wireless friend, Woohookitty, Xphile2868, Xxsquishyxx, Zondor, ¦§• •¨•®ª¡ ¡¢«¢¬§¢, ˜ -®ª«¢ž ¯®•°©•£, 152 anonymous edits

Java Database Connectivity *[Source*: http://en.wikipedia.org/w/index.php?oldid=576879686 *[Contributors*: AS, Abdull, Andreas Kaufmann, Andrewman327, Arcann, Atozjava, Audriusa, Beetstra, Bkrakesh, Blaine-dev, Brick Thrower, Bunnyhop11, CambridgeBayWeather, Cameltrader, Cander0000, Caomhin, Captmjc, Cherkash, Chowbok, ChrisGualtieri, Chrismacgr, Coldacid, Danim, Darth Molo, Dddelfin, Derbeth, Dinosaurdarrell, DivideByZero14, Doug Bell, Drewgut, Edward, Ehheh, Ehn, Elandon, Elaz85, Eumolpo, Evgeni Sergeev, Explanator, FOOL, Faisal.akeel, Ferengi, Ferrans, Fikus, Forage, Frecklefoot, Fred Bradstadt, Fuhghettaboutit, Fyyer, GLumba, GraemeL, Graham87, GreyCat, Harryboyles, Iain.dalton, Insomnia64, JDvorak, JLaTondre, Jacks435, Jameboy, Jay, Jems421, JeremyStein, Jeronimo, Jjaazz, Joffeloff, Josepant, Josephw, Julesd, Katieh5584, Kaydell, KingsleyIdehen, Klausness, Lenaic, Leszek4444, M4design, M4gnum0n, MER-C, MacTed, Mark Renier, Materialscientist, MeltBanana, Mindmatrix, Mintleaf, Moribunt, MrOllie, Nlevitt, Noq, Pako, Pillefj, Pinecar, Pinkyf, Piojo, Pne, Poor Yorick, Praslisa, Ravenmewtwo, RedWolf, Reedy, Rm1507, SJK, Sae1962, Schw3rt, Sfmontyo, ShaunMacPherson, Simran.preet90, Sjc, SmackEater, Sohail.sikora, Sqlinfo, Srecd, Sualeh, Supreme geek overlord, Tcncv, The Anome, The Aviv, Theresa knott, Tom 99, Twsx, Unclejedd, Warren, Wikipelli, Winterst, Yaniv Kunda, Yaxh, Zundark, Ÿª±°¢ –ª°©-²¢ž-³, Ž • • , 219 anonymous edits

XQuery API for Java *[Source*: http://en.wikipedia.org/w/index.php?oldid=572931513 *[Contributors*: A5b, Bgwhite, Bxj, Codename Lisa, F331491, Frap, Gurt Posh, Klemen Kocjancic, Mhkay, Protonk, RHaworth, Vegaswikian, Yutsi, 23 anonymous edits

ODBC *[Source*: http://en.wikipedia.org/w/index.php?oldid=566327529 *[Contributors*: AKGhetto, AdventurousSquirrel, AlistairMcMillan, Allens, Andreas Kaufmann, AndriuZ, Andy Dingley, Arch dude, Auric, AvicAWB, Av…, Beevvy, Bigpru, BobGibson, BonsaiViking, Borgx, Bovineone, BryEllis, Bunnyhop11, Cander0000, CanisRufus, Canterbury Tail, Charlesgold, Chealer, ClaudioSantos, Computafreak, Craig Stuntz, DFulbright, Danim, David Gerard, Derbeth, Discospinster, Dittaeva, DragonHawk, Drewgut, Eglobe55, Electrolite, Epim, Everlasting Winner, Gcm, GreyCat, Gwern, Harry Wood, Inzy, Irish all the way, JLaTondre, Jandalhandler, Jay, Jerome Charles Potts, Jkelly, Jklowden, John of Reading, JonathanMonroe, KeyStroke, KingsleyIdehen, Kuru, Kyouteki, Kzafer, Larsinio, Lkstrand, Lowellian, Lurcher300b, MacTed, Magnus.de, Manop, Mark Renier, Markhurd, Martijn Hoekstra, Materialscientist, Maury Markowitz, Maximaximax, MeltBanana, Michael Hardy, Mikeblas, Mindmatrix, Minesweeper, Minghong, Mintleaf, Misog, Mitsukai, NapoliRoma, Nikos 1993, Nixdorf, NoahSussman, Not Sure, Orlady, Orpheus, Oxda, Pajz, Paul Foxworthy, Pedant17, Pmsyyz, Polluks, Power piglet, PrisonerOfIce, Quuxa, Raffaele Megabyte, Rajkumar9795, Reconsider the static, Reedy, RenniePet, Rjwilmsi, Rrabins, Seanwong, Sega381, Spellmaster, Sspecter, Struway, Swhalen, The Anome, The wub, Thumperward, Tide rolls, Tin, Todorojo, TommyG, Viridae, Wez, Whpq, Wikipelli, William Avery, Winterst, Woohookitty, Ysangkok, Yug1rt, ´ªµ-¶±·¸§¸, 231 anonymous edits

Query language *[Source*: http://en.wikipedia.org/w/index.php?oldid=575926344 *[Contributors*: ASHPvanRenssen, Adarw, Ahoerstemeier, Ahunt, Albert688, Amin Hashem, AmirMehri, AndrewWarden, BCable, Bacchus123, Beaton1131, BenAveling, Bkonrad, Chtirrell, CxQL, DEng, Danakil, Danim, Davidfstr, Deepugn, Devourer09, Diamondland, ERfan111, Edward, Ehajiyev, Elwikipedista, Face, Frieda, Groovenstein, Grutness, HanielBarbosa, Honys, Ihenriksen, Inverse.chi, IvanLanin, Jay42, Joerg Kurt Wegner, John Vandenberg, John of Reading, Jonathan.mark.lingard, KeyStroke, Kwiki, Larsinio, Logiphile, MarXidad, Mark Arsten, Mark Renier, Markhobley, Mgreenbe, Mhkay, MichaelSpeer, Mild Bill Hiccup, Msnicki, NGC 2736, Nikola Smolenski, Ojigiri, OsamaK, Peter Gulutzan, Retireduser1111, Rfl, SarekOfVulcan, Shekhardtu, Slipstream, Soumyasch, Srandrews, Steven Walling, Svick, Tassedethe, Techno.modus,

Throbblefoot, TommyG, Toussaint, Trevor MacInnis, Troels Arvin, Usien6, Valafar, Vanished user qkqknjitkcse45u3, Vmenkov, Wikiolap, Xodlop, ZygmuntKrynicki, 48 anonymous edits

Query optimization *[Source*: http://en.wikipedia.org/w/index.php?oldid=574425789 *[Contributors*: Abdull, Andreas Kaufmann, Andy Dingley, Avalon, Bearcat, Beland, Cadvga, Cedar101, Danim, Edward, Ginsuloft, Glux, GregorB, Gzuckier, Isulica, JoshRosen, Less Than Free, MBisanz, Mild Bill Hiccup, Mouchoir le Souris, MrOllie, Mrmatiko, Nadeemhussain, Neilc, Owl3638, Paige Master, Pascal.Tesson, Ronwarshawsky, Sct72, Sudhir h, TechPurism, Vitriden, Walter G¹rlitz, 24 anonymous edits

Query plan *[Source*: http://en.wikipedia.org/w/index.php?oldid=574422744 *[Contributors*: Aaronbrick, Alpha Quadrant, Ammar.w, Ancheta Wis, Arcann, Bevo, Cedar101, Cww, Freezegravity, Grace Note, Hardeeps, James barton, Larsinio, Mark Renier, Mbarbier, Mdesmet, Mikeblas, Mindmatrix, Neilc, Nikola Smolenski, Reedy, RI, Ronwarshawsky, SimonP, Sippsin, Slaniel, TheParanoidOne, UnitedStatesian, Walter G¹rlitz, Woodshed, ZoBlitz, 26 anonymous edits

Database administration and automation *[Source*: http://en.wikipedia.org/w/index.php?oldid=575478582 *[Contributors*: Aflorin27, Akerans, Anas2048, Beetstra, Cuttysc, Dabron, David.lamberth, Dr Gangrene, Drunkenmonkey, Elonka, Ericgross, Hffmgb899, ITautomationFreak, JEH, JaGa, Jamesx12345, Jpbowen, Kjkolb, Kku, Kukushk, MZMcBride, Maahela, Mwtoews, Oli Filth, Piano non troppo, Pointillist, Qwyrxian, R'n'B, Ronz, Rwwww, Rybec, ShelfSkewed, The Thing That Should Not Be, Theopolisme, Vegaswikian, 64 anonymous edits

Replication (computing) *[Source*: http://en.wikipedia.org/w/index.php?oldid=577000800 *[Contributors*: Artlung, Ashutosh y0078, Autiger, Beland, Bloodshedder, Bsilverthorn, Bsoo, Daniel7066, David--swenson, Dimos2k, DixonD, Elhaddi, Elsendero, Fabkins, Fairwin99, GFHandel, Giteshtrivedi, Grafen, Hgabri1, Holiwud111, Ian wild, Indeyets, Intgr, Ispabierto, J.delanoy, Jaksa, Jamelan, Jay, John of Reading, JonHarder, Jsaylor3, Keithathaide, Ken Birman, Klightspeed, Klilidiplomus, Kubanczyk, Kwiki, Leroyvl, Manceraa, MarkusSchiltknecht, Maximaximax, Mercy11, Mihao2am, Mion, Miym, Momo54, Neilc, NicDumZ, Ohnoitsjamie, Owlsfan, Philipcurrito, Pmedema, Pnm, Psustman, RShohat, Radagast83, Reedy, Ronz, Rror, Ruralhouse, ScottEvonLewis, Sehwag, Someguy1221, Speculos, TedDunning, Tedickey, TheParanoidOne, Tobias Bergemann, Twimoki, Vincefleming, Wbm1058, Wilee, Winterst, X7q, 104 ,°»½Ù½¼ ¿à¾Á anonymous edits

Comparison of object database management systems *[Source*: http://en.wikipedia.org/w/index.php?oldid=568898654 *[Contributors*: Alexandre.Morgaut, Bablind, Beland, Brunov, CWuestefeld, ChrisGualtieri, Cristiursachi, D aana, Dmytrob, Espresso999, FatalError, Ftiercel, George A. M., Harryboyles, Hu12, Hyspdrt, JGrosmann, JLaTondre, Jarble, Jerome Charles Potts, Kiore, Knowlengr, Lguzenda, MMSequeira, MacTed, Manfred-jeu, Matspca, Minas.w, Nihiltres, Palosirkka, Pwaddles, R'n'B, Radu124, Rjolly, Sadads, Siaqodb, SpeoLeo, Spolnik, Svetoslav.Mateev, Talyian, Tekktura, The Founders Intent, Thegreeneman5, Torc2, Uncommon Sense, Woohookitty, Xiloynaha, 109 anonymous edits

Comparison of object-relational database management systems *[Source*: http://en.wikipedia.org/w/index.php?oldid=566897826 *[Contributors*: Akagel, Alexandre.Morgaut, Anas2048, Beland, Beta m, Calabrese, Chikako, Chris the speller, Christian75, Cigano, Cubridorg, DRady, Donhalcon, Garyzx, Ghp, Gudeldar, JJay, Jeff3000, Jerome Charles Potts, Karnesky, KingsleyIdehen, Leotohill, Lotje, MER-C, Mark Renier, Minghong, Palosirkka, Pamri, Piano non troppo, Reedy, Requestion, Ruud Koot, Rwwww, Salix alba, Skyezx, Squids and Chips, Versus22, Wmahan, 19 anonymous edits

List of relational database management systems *[Source*: http://en.wikipedia.org/w/index.php?oldid=572498598 *[Contributors*: *drew, 2A00:FE00:BFFE:2201:0:0:0:400, AaaghItsMrHell, Adamblang, Adrian J. Hunter, AlistairMcMillan, Alureiter, Amalthea, Anas2048, Arcann, Armadillo ECM, Armen1304, Baojia, Basil.bourque, Beland, BoxSoft, Bp0, Bressan, CasperGoodwood, Cburnett, Cgfdmc, ChandraASGI, CharlieTesta*24, Closedmouth, Countersubject, CovenantD, Craig Stuntz, Crosbiesmith, Cubridorg, DEddy, DRady, Darren Duncan, DatabACE, Davidsheiman, Dbaxter0, DerHexer, Dfetter, Dougbertram, Eeekster, Einarkristjan, Ekraft, Elf, Fraise, Fschupp, Ggeldenhuys, Glange90411, Greenrd, Grstain, Gsingh, HJ Mitchell, HappyCamper, Herostratus, Hertzsprung, Igloobone, Imroy, Jam02, JasonThePirate, Jberkus, Jerome Charles Potts, JethroElfman, Jimgawn, Jimmi Hugh, JohnGray, JohnnyMrNinja, Jonasfagundes, Jost Riedel, Jpetersen74, Jtdunlop, Karnesky, KingsleyIdehen, Kognitio, Kriplozoik, Leandrod, Legolas558, Lfstevens, Lowellian, Lpetrazickis, Luke Lonergan, MainFrame, Mark Renier, Markoprima, MarylandArtLover, Mcaisse, Mekong Bluesman, Mikeblas, Mindmatrix, Minghong, MrBoo, Mrozlog, Mtasic, Neilrieck, Ngpd, Nick Number, Nicolas1981, Niteowlneils, Ocherkashin, OlivierWeb, Paul A, Pelagius333, Pengo, Pgillman, Pjrm, Prolog, Pwinkler4185, Radagast83, Ramanna.Sathyanarayana, Rcorcs, Reedy, Requestion, Rich Farmbrough, Ross Burgess, Rrabins, Ryandaum, Sappy, ScottDavis, Sergsav, Snarius, Sqlinfo, StevenBlack, Stuboy, Sushi500, Syrthiss, Ted nw, Tehnic49, Tentinator, Thomashilbert, Thryduulf, TommyG, Troels Arvin, Tstevelt, Turnstep, Vmatikov, Vtbl, W163, WOSlinker, Waw2010, Whouk, Wikiolap, Williamtimm, Zimbabwer, 177 anonymous edits

Comparison of relational database management systems *[Source*: http://en.wikipedia.org/w/index.php?oldid=577589864 *[Contributors*: 2A00:FE00:BFFE:2201:0:0:0:300, 2A00:FE00:BFFE:2201:0:0:0:400, 90, 96cores, A5b, Abu badali, Adono, Aeriform, Agavenwurm, Akagel, Alexandre.Morgaut, Alvaro.Monge, Analoguedragon, Anas2048, AndrewCowie, Angoca, Astral v, Axelstudios, Az29, BBCWatcher, Basil.bourque, Beetstra, Beland, Beta m, Bezik34, Bgwhite, Bigown, Bjkeefe, BlackCatN, Bmfrosty, Bogdangiusca, Brick Thrower, Brierand, Brookie, Brulath, CCFS, CRAIZ3D, Calador109, Carp3, Cazito, Ceaton55, Chachka, Chendy, Cheolsoo, Chris the speller, Chriskl, Cjcollier, Clanie, ClementSeveillac, Clieu, Comp.arch, Coolboy1234, Corrado, CovenantD, Craig Stuntz, Cubridorg, Curps, DanBishop, DancingMan, Danmcg.au, Dark ixion, Darthsco, DavidMCEddy, Davidsheiman, DeTru711, DeirdreGerhardt, Dfetter, Dionyziz, DocendoDiscimus, Donhalcon, Dougdp, DrThompson, Drilnoth, Duckbill, Dveeden, Dvgeorge, Ean5533, Edward, Eli lilly, Equalizer777, EvanCarroll, Findling67, Fivelittlemonkeys, Fixesfixes, Florian Sening, Fonsie, Frap, Fratrep, Fredrik, Gbgsimulationjon, Gcalis, Gczffl, Geordee, Georgeryp, Gilad.maayan, Gintsp, Gkanel, Glange90411, Glmeece, Gmaxwell, Goeldner, Gogo Dodo, Gongshow, Greenman, GreyCat, Gudeldar, Guerrabraga, HHempelmann, HappyCamper, HarrisonFisk, Hasegeli, Hibethy, Hrgwea, Hu12, Hz.tiang, Igloobone, Improv, Imroy, Indexheavy, Inessa4ever, Intgr, Isaac Sanolnacov, JJay, JLaTondre, Jamesday, Jawsper, Jberkus, Jbicik, Jeltz, Jerome Charles Potts, Jethro555, Jevansen, Jhonjairoroa87, Jim.Callahan,Orlando, Jin.Takahashi, Jmachat, Jmnbatista, Jojalozzo, Jon207, Jot1109, Jpupier, Judyburk, Juha001, Kadishmal, Kaelfischer, Karnesky, Kayvee, Keldar, Kempeth, Kenfar, Kent Heiner, KingsleyIdehen, KiwiBiggles, Kmorozov, Kognitio, Kozmando, Kweetal, Kweetal nl, Kytti khat, LHCgrp, LarsHolmberg, Larsinio, Latios, Leandrod, Leandrpf, Lfstevens, Lguzenda, LiX, Lightblade, Lmxspice, Lowellian, Lowmeus, Lucas Malor, Lzcubs1, M Th vat, Maarten Hermans, MacTed, Maheshgadgil, Manifoldtop, Maple10, Mariuz, Mark Renier, Marklark, MeekMark, Mhagman, Mikeblas, Mikluce, Mindmatrix, Minghong, Misery, Mo ainm, Moocha, Moralis, MrBoo, Mtasic, Mwtoews, Mywikie, Måthrandir, NaibStilgar, NapoliRoma, Naraht, Narendra Sisodiya, Naviworx, Ne0Freedom, Neilc, Ngpd, Nhantdn, Nickdc, Niqueco, Niteowlneils, Nodulation, Noonand, Noxia, OKIsItJustMe, Odinblade, Ofbarea, Olivier Debre, PCJockey, Patheticcockroach, PentoMcGreno, Peterl, Petri Krohn, Pgan002, PhilHorder, PieterDeBruijn, Plesatejvlk, Plumcreek, Pnv82, Proofreader77, Pvjohnson, Pwsegal, Quebec99, Radagast83, Radio15dude, RandalSchwartz, RaniaSOUSSI, Reedy, Reisio, Rhaas, Rhobite, Rich Farmbrough, RickBeton, Rimio, RjwiImsi, Robert K S, Rsocol, Rudi.Leibbrandt, Rupert160, Ruud Koot, Ryguasu, S.K., SDSWIKI, SLi, Seashorewiki, Sehbueno, Shenme, Shusseina, Sietse Snel, SixSkys, Slaweks, Slyzius, Snarius, Snarpel, Soeren1611, Sqlboy, Sqlinfo, Stephenw32768, Stuboy, TRauMa, Tabletop, Taichi, TallMagic, Tarjei Knapstad, Taylorsharpemac, TechPurism, Tedmcneal, Terfilo, Tfischbeck, Tharakan, The Anome, ThomasMueller, Thunderbird2, Thunderbritches, TimTay, Timwi, Tlaresch, TobiasPersson, TommyG, Tranemonet, Troels Arvin, Turanyuksel, Turnstep, Udittmer, Ukuechle, Unicard-ic, Veliscu Ovidiu, Victorwss, Vincenzo.romano, Vrenator, W3bbo, WOSlinker, Waldir, Waveform, Waw2010, Waynelwarren, WereSpielChequers, Whimsley, Wielewaal, Wikirosi, Wild Pansy, Will henderson, William Avery, Williamtimm, Wiml, Wmahan, Woohookitty, Wtuvell, Xpclient, Xprotocol, Yourbane, Yukuku, Yzchang, Zero0w, Zollhausring, Zsoltika, – — ˜, 927 anonymous edits

Document-oriented database *[Source*: http://en.wikipedia.org/w/index.php?oldid=574142205 *[Contributors*: Altered Walter, Antony.stubbs, Argv0, Arthena, Bablind, BrideOfKripkenstein, Bunnyhop11, Bxj, CJGarner, Cbuccella, Cedar101, Chris Wood, ChristianGruen, Cobaltbluetony, Compfreak7, Crosstantine, Cybercobra, Danim, Danmcg.au, Dasfrosty, Dm, Dmccreary, Dodilp, Edward, Ehn, Enric Naval, EricBloch, FatalError, Frap, FreeRangeFrog, Goldzahn, Gwicke, Hzguo, Imroy, Iznogoud, JIP, Jerome Charles Potts, Jwoodger, Kingsleyj, Kirt, Lodrian, Mark Arsten, Mark Renier, Mbroberg, Mdd, Mindmatrix, Mortense, Mqchen, MySchizoBuddy, Neitherk, Niceguyedc, Nikhil Umesh, Nwbeeson, Pcap, Philu, Plamoa, Plasma east, Pointillist, QuiteUnusual, R'n'B, Rachkovsky, Rediosoft, Refactored, Rfl, Rtweed1955, SDSWIKI, Spdegabrielle, Stuartyeates, Superjordo, Thorwald, Thumperward, Toutoune25, Tsm32, Woohookitty, 105 anonymous edits

Graph database *[Source*: http://en.wikipedia.org/w/index.php?oldid=577575136 *[Contributors*: 0x24a537r9, 2001:18E8:2:1031:50C:3655:857A:F9BB, 2001:818:D916:2200:61FD:4BE7:E6C4:2855, 2001:858:5:201:65C0:B01F:1C0D:B10E, 2A01:E35:2E41:7720:9D1:92AE:1786:6B9B, 4368a, Agavenwurm, Aglnl, Ahzf, Ajmagnifico, Aldonline, Andrearatto, Bolerio, Bunnyhop11, Cnorvell, Colinniu, Crcsmnky, DamarisC, Danim, Dreamingxk89, E40, Egbert J. van der Haring, Electro rick, Elykahn1, Espeed, Fceller, Ffangs, Fraktalek, Frap, Freshnfruity, Germanviscuso, Giftlite, J12t, JakobVoss, Jmesney, Jncraton, Jni, Jonik, Kiryakov ak, Ksinkar, Lambdazen, Lesser Cartographies, Lguzenda, Lillem4n, Luebbert42, Luisbargu, MacTed, Magnuschr, Materialscientist, Miami33139, Michael A. White, MoSarwat, Morphh, MuffledThud, Nawroth, Pelister, Pereb, Pholding, Pointillist, Ppr15, Praveensripati, ProfessorBaltasar, RecaiAlkan, RichMorin, SamJohnston, Sbrunner, Shengqiyang, Starboy8, Stott.parker, Stybn, Syhuang1988, TTJDenman, TempestSA, Tgrota, Thinxer, Thomas888b, Thoughtpuzzle, Tsm32, Tuhl, Wbeaureg, Yanivby, 126 anonymous edits

NoSQL *[Source*: http://en.wikipedia.org/w/index.php?oldid=577845207 *[Contributors*: (:Julien:), 2001:980:259E:1:B121:272F:7400:3B6D, 2A02:810D:12C0:4B:5C1A:6342:A931:F78B, Adtadt, Al3xpopescu, Alexandre.Morgaut, Alexrakia, AlisonW, Altered Walter, Amire80, Anastrophe, AndrewBass, Angry bee, Anilkumar1129, Anoop K Nayak, Ansh.prat, Anujsahni, Argv0, Arjayay, Arto B, Asafdapper, Ashtango, Ashtango5, Atropos235, AxelBoldt, BD2412, Bbulkow, Bdijkstra, Bearcat, Beland, Benatkin, Benhoyt, Bhaskar, Billinghurst, Biofinderplus, Boshomi, Bovineone, Bramante, Brocsima, CJGarner, CaptTofu, Ceefour, Cekli829, Charbelgereige, Chenopodiaceous, ChristianGruen, Ciges, Clemwang, Cloud-dev, Cnorvell, ColdShine, Coldacid, Compfreak7, Corrector623, Craigbeveridge, Crosbiesmith, Crosstantine, Cybercobra, Cyril.wack, DBigXray, Dabron, DamarisC, Dancrumb, DatabACE, DavidBourguignon, DavidSol, Davidhorman, Dawn Bard, Dericofilho, Dewritech, Dm, Dmccreary, Dmitri.grigoriev, Dredwolff, Drttm, Dshelby, Dstainer, Duncan, Ebalter, Eco schranzer, Edlich, Ehn, Electricmuffin11, Eno, EricBloch, ErikHaugen, Ertugka, Euphoria, Excirial, Extrovrt101, F331491, Farvartish, Fiskbil, Fitzchak, Fmorstatter, FontOfSomeKnowledge, Fraktalek, FranzKraun, Frap, Freshnfruity, Frze, Furrykef, Fxsjy, Gaborcselle, Gadfium, Germanviscuso, Getmoreatp, GimliDotNet, Ginsuloft, Gkorland, GlobalsDB, GoingBatty, Gonim, Gorman, Gpierre, GraemeL, Griswolf, Gstein, Hairy Dude, Harpreet dandeon, Headbomb, Heelmijnlevenlang, Hloeung, Hoelzro, Hu12, Inmortalnet, Intgr, Irmatov, JLaTondre, Jabawack81, Jandalhandler, Jasonhpang, Javalangstring, Jeffdexter77, Jerome Charles Potts, JnRouvignac, Jnaranjo86, JohnPritchard, Jonasfagundes, Joolean, Jottinger, Jrudisin, Jstplace, Jubalkessler, Justinsheehy, Kbrose, Kgfleischmann, Khiladi 2010, Ki2010, KiloByte, Kkbhumana, Kku, Knudmoeller, Koavf, Komap, Korrawit, LeeAMitchell, Leegee23, Legacypath, Leotohill, Lfstevens, Lguzenda, Linas, Lmxspice, Looris, Luebbert42, Luisramos22, Lyoshenka, MMSequeira, Mabdul, MacTed, Magnuschr, ManikSurtani, Marasmusine, Mark Arsten, Matspca, Matt Heard, Mauls, Mauro Bieg, Maury Markowitz, Mbarrenecheajr, Mbonaci, Mbroberg, McSly, Mesut.ayata, Mhegi, Miami33139, Mitpradeep, Mjresin, Morphh, Mortense, MrOllie, MrWerewolf, Msalvadores, Mshefer, Mtrencseni, Mydoghasworms, Nanolat,

Natishalom, Nawk, Nawroth, Netmesh, Neustradamus, Nick Number, Nileshbansal, Nosql.analyst, NtoII, OmerMor, Omidnoorani, Orenfalkowitz, Ostrolphant, PatrickFisher, Pcap, Peak, Pereb, Peter Gulutzan, Phillips-Martin, Philu, Phoe6, Phoenix720, Phunehehe, Plustgarten, Pnm, Poohneat, ProfessorBaltasar, QuiteUnusual, Qwertyus, R39132, RA0808, Rabihnassar, Raysonho, Razorflame, Really Enthusiastic, Rediosoft, Rfl, Robert1947, RobertG, Robhughadams, Ronz, Rossturk, Rpk512, Rtweed1955, Russss, Rzicari, Sae1962, Sagarjhobalia, SamJohnston, Sandy.toast, Sanspeur, Sasindar, ScottConroy, Sdrkyj, Sduplooy, Seancribbs, Seraphimblade, Shadowjams, Shepard, Shijucv, Smyth, Socialuser, Somewherepurple, Sorenriise, Sstrader, StanContributor, Stephen Bain, Stephen E Browne, Steve03Mills, Stevedekorte, Stevenguttman, Stimpy77, Strait, Syaskin, TJRC, Tabletop, Tagishsimon, Techsaint, Tedder, Tgrall, The-verver, Theandrewdavis, Thegreeneman5, Thomas.uhl, ThomasMueller, Thumperward, ThurnerRupert, Th‘ringer, Timwi, Tobiasivarsson, Tomdo08, Trbdavies, Tshanky, Tsm32, Tsvljuchsh, Tuvrotya, Tylerskf, Ugurbost, Uhbif19, Vegaswikian, Violaaa, Viper007Bond, Volt42, Voodootikigod, Vychtrle, Walter G¹rlitz, Wavelength, Weimanm, Whimsley, White gecko, Whooym, William greenly, Winston Chuen-Shih Yang, Winterst, Woohookitty, Wyverald, Xtremejames183, YPavan, Zapher67, Zaxius, Zond, – -®ª§ Ã•®-®ªž¬-³, 654 anonymous edits

NewSQL *¡Source*: http://en.wikipedia.org/w/index.php?oldid=577827531 *¡Contributors*: Akim.demaille, Amux, Apavlo, Beland, Brianna.galloway, Diego diaz espinoza, Ibains, Intgr, Julian Mehnle, MPH007, MacTed, Maury Markowitz, MrOllie, Mwaci99, Plothridge, Quuxplusone, Stuartyeates, UMD-Database, 12 anonymous edits

Image Sources, Licenses and Contributors

File:CodasylB.png *{Source*: http://en.wikipedia.org/w/index.php?title=File:CodasylB.png *{License*: Creative Commons Attribution-ShareAlike 3.0 Unported *{Contributors*: Jean-Baptiste Waldner, User:Jbw

Image:Relational key.png *{Source*: http://en.wikipedia.org/w/index.php?title=File:Relational_key.png *{License*: Public Domain *{Contributors*: LionKimbro

File:Database models.jpg *{Source*: http://en.wikipedia.org/w/index.php?title=File:Database_models.jpg *{License*: Creative Commons Attribution-Sharealike 3.0 *{Contributors*: Marcel Douwe Dekker

Image:A2 2 Traditional View of Data.jpg *{Source*: http://en.wikipedia.org/w/index.php?title=File:A2_2_Traditional_View_of_Data.jpg *{License*: Public Domain *{Contributors*: itl.nist.gov

Image:Flat File Model.svg *{Source*: http://en.wikipedia.org/w/index.php?title=File:Flat_File_Model.svg *{License*: Public Domain *{Contributors*: Wgabrie (talk) 16:48, 13 March 2009 (UTC)

Image:Hierarchical Model.svg *{Source*: http://en.wikipedia.org/w/index.php?title=File:Hierarchical_Model.svg *{License*: Public Domain *{Contributors*: U.S. Department of Transportation vectorization:

Image:Network Model.svg *{Source*: http://en.wikipedia.org/w/index.php?title=File:Network_Model.svg *{License*: Public Domain *{Contributors*: U.S. Department of Transportation vectorization:

File:Emp Tables (Database).PNG *{Source*: http://en.wikipedia.org/w/index.php?title=File:Emp_Tables_(Database).PNG *{License*: Public Domain *{Contributors*: Jamesssss

Image:Object-Oriented Model.svg *{Source*: http://en.wikipedia.org/w/index.php?title=File:Object-Oriented_Model.svg *{License*: Public Domain *{Contributors*: U.S. Department of Transportation vectorization:

File:Update anomaly.svg *{Source*: http://en.wikipedia.org/w/index.php?title=File:Update_anomaly.svg *{License*: Public Domain *{Contributors*: Nabav,

File:Insertion anomaly.svg *{Source*: http://en.wikipedia.org/w/index.php?title=File:Insertion_anomaly.svg *{License*: Public domain *{Contributors*: en:User:Nabav, User:Stannered

File:Deletion anomaly.svg *{Source*: http://en.wikipedia.org/w/index.php?title=File:Deletion_anomaly.svg *{License*: Public domain *{Contributors*: en:User:Nabav, User:Stannered

File:Referential integrity broken.png *{Source*: http://en.wikipedia.org/w/index.php?title=File:Referential_integrity_broken.png *{License*: GNU Free Documentation License *{Contributors*: en:User:Ta bu shi da yu

Image:Relational database terms.svg *{Source*: http://en.wikipedia.org/w/index.php?title=File:Relational_database_terms.svg *{License*: Public Domain *{Contributors*: User:Booyabazooka

File:Relational Model.svg *{Source*: http://en.wikipedia.org/w/index.php?title=File:Relational_Model.svg *{License*: Public Domain *{Contributors*: U.S. Department of Transportation vectorization:

File:Relational key.png *{Source*: http://en.wikipedia.org/w/index.php?title=File:Relational_key.png *{License*: Public Domain *{Contributors*: LionKimbro

File:Relational model concepts.png *{Source*: http://en.wikipedia.org/w/index.php?title=File:Relational_model_concepts.png *{License*: GNU Free Documentation License *{Contributors*: User:AutumnSnow

File:Object-Oriented Model.svg *{Source*: http://en.wikipedia.org/w/index.php?title=File:Object-Oriented_Model.svg *{License*: Public Domain *{Contributors*: U.S. Department of Transportation vectorization:

File:Db null.png *{Source*: http://en.wikipedia.org/w/index.php?title=File:Db_null.png *{License*: GNU Free Documentation License *{Contributors*: User:SqlPac

File:XQJ-Architecture.svg *{Source*: http://en.wikipedia.org/w/index.php?title=File:XQJ-Architecture.svg *{License*: GNU Free Documentation License *{Contributors*: F331491

File:Storage replication.png *{Source*: http://en.wikipedia.org/w/index.php?title=File:Storage_replication.png *{License*: Creative Commons Attribution-Sharealike 3.0 *{Contributors*: User:Speculos

File:GraphDatabase PropertyGraph.png *{Source*: http://en.wikipedia.org/w/index.php?title=File:GraphDatabase_PropertyGraph.png *{License*: Creative Commons Zero *{Contributors*: User:Obersachse

License

Creative Commons Attribution-Share Alike 3.0
//creativecommons.org/licenses/by-sa/3.0/