Bulletin of the Technical Committee on

Data Engineering

September 2007 Vol. 30 No. 3 **IEEE Computer Society**

Letters

Special Issue on Embedded and Mobile Database Systems

Conference and Journal Notices

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

There are two Data Engineering Bulletin web sites: http://www.research.microsoft.com/research/db/debull and http://sites.computer.org/debull/.
The TC on Data Engineering web page is http://www.ipsi.fraunhofer.de/tcde/.

Letter from the Editor-in-Chief

The Current Issue

Database technology is now ubiquitous. Indeed, it is increasingly hard to avoid. But it is no longer restricted to "computers". Indeed, I would not be surprised to find a database as part of my toaster at some point. Databases are already a part of cell phones, music players, TV set top boxes, and just about everything else within the domain of consumer electronics. There are a couple of fascinating aspects of this database triumph.

1. A user does not "see" a database. Rather, the database is completely embedded within an application and its functionality is accessible via a user interface that is specialized to the task at hand, e.g. storing and retrieving songs in a portable music player.

2. The database is frequently transported, within a portable device. This requires that the database system have a much smaller code footprint than we have come to expect in our computer servers. It also means that the database system must be prepared to deal with flash storage and take power dissappation into account.

Both the above considerations change in dramatic ways, the implementation of database systems.

1. The fact that the user does not see the database means that the database interface need not be SQL. Indeed, it need not be relational, and it might have rather circumscribed functionality. This can make the query processing task and the related physical database design task more simple and enhance the self-managability of the system.

2. The new portable platforms present a new set of hardware architectures and hence a new set of design considerations in the building of a database system. For example, if flash memory is used, then the database system implementer needs to understand the write/erase characteristics of flash, and design the appropriate storage structures.

Anastasia Ailamaki has assembled the current issue of the Bulletin on the subject of "Embedded and Mobile Database Systems". In the tradition of the Bulletin, the issue contains an interesting mix of articles from both university researchers and industrial database specialists. The issue both describes this exciting area and sheds light on interesting aspects of how new database systems are dealing with what is both an opportunity and a challenge. I want to thank Natassa for her fine job as editor of this issue, which should be interesting reading for many in our community.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

With the recent advent of mobile computing devices such as PDAs, cellular phones, sensors, and smart cards, data management on such devices has become an important research area. Interestingly, very small data bases are just as challenging to manage as very large data bases, because the traditional large database systems designed to manipulate terabytes of storage on multiprocessor engines are not appropriate for small devices. The difference lies in three major factors: First, resources in tiny devices are typically scarse, therefore designers of embedded database systems must minimize footprint, memory references, CPU cycles, network bandwidth, and more importantly, consumption of battery power. Second, mobility makes cache consistency a much more challenging problem than it is in traditional distributed client/server systems. Third, the software needs to be robust and, once installed, should function without needs for administration.

This special edition of the IEEE Data engineering Bulletin is devoted to Embedded and Mobile Database Systems, and includes a representative collection of articles which describe efforts to address important problems in the field. The issue opens with Anil Nori's survey of the problems and current efforts in the area, followed by a more detailed study on caching and replication for mobile databases by Panos Chrysanthis and Evi Pitoura. The challenges of embeddability and their solutions are detailed in the next three articles: Margo Seltzer contributed an article on the ubiquitous BerkeleyDB, now part of Oracle, but traditionally part of numerous research projects as a versatile embedded database system. An article on the SQL Anywhere server by Sybase iAnywhere follows, which explains the main design challenges and ideas of this popular embedded commercial product. In addition, the inspirers of TinyDB – the must-have to successfully and reliably collect and query data on dynamic, ever-growing sensor networks – sent us an article describing the main aspects of the system and related reserch efforts by the same group. The issue concludes with an article by Anciaux et. al. at INRIA, which explains the challenges and trends when managing data on secure chips, such as smart cards.

I would like to cordially thank the authors of the articles above, who graciously devoted their time and effort to contribute articles to this special edition. I hope that you will enjoy reading it.

Anastasia Ailamaki
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
Ecole Polytechnique Fédérale de Lausanne
Lausanne, Switzerland

Mobile and Embedded Databases

Anil K. Nori
Microsoft Corporation
One Microsoft Way, Redmond, WA 98052
anilnori@microsoft.com

Abstract

Recent advances in device technology and connectivity have paved the way for next generation applications that are data-driven, where data can reside anywhere, can be accessed at any time, from any client. Also, advances in memory technology are increasing the capacities of RAM and Flash, and their costs down. These trends lead to applications that are mobile, embedded, and data-centric. This paper presents an overview of mobile and embedded database systems and applications.

1 The New Environment - Mobile and Embedded

Recent advances in processors, memory, storage, and connectivity have paved the way for next generation applications that are data-driven, whose data can reside anywhere (i.e. on the server, desktop, devices, embedded in applications) and that support access from anywhere (i.e. local, remote, over the network, in connected and disconnected fashion). Memory sizes have gone up and prices have come down significantly; with 64 bit addressability, it is not uncommon to configure servers with 8 – 16GB of memory, and desktops with 2 – 4 GBs of memory. With advances in flash memory technology, large flash drives are available at reasonable prices. Computers with 32 GB flash drives are making way into the market. Flash drives not only eliminate seek time and rotational latency they consume significantly less power than conventional disk drives, making them ideal for portable devices. All this naturally leads to mobile, disconnected, and specialized application architectures. Application components can run on different tiers, in different service (autonomy) boundaries, and on different platforms (e.g. server, desktop, mobile). These new breeds of applications fall into one or more of the following categories:

1.1 Mobile

As more users adopt Wi-Fi enabled laptops, and with increasingly capable mobile devices, the need for mobile applications is increasing. Applications like Email, Calendaring, CRM (Customer Relationship Management) are already targeting mobile devices. Middleware infrastructures like application servers and workflow services are becoming mobile-aware. Some reasons for such mobility trends are:

- More employees are mobile. Email and offline access is becoming pervasive.
- Mobile usage is broadening. Mobile usage is already prevalent in certain vertical domains like Healthcare, Insurance, and Field Services.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

- Mobile applications are more than just browser windows – more and more applications now run natively on mobile devices.

Data management and access on mobile devices is central to mobile applications. As mobile applications achieve widespread adoption in the enterprise, mobile and embedded DBMSs – needed to support such applications become an important part of the IT infrastructure. And as these applications grow more disconnected and sophisticated with increasing data sizes the need for rich data processing capabilities increases.

We also note that consumer and home user applications are becoming web-centric and data is being stored on the web (cloud) and accessed from anywhere, at any time. Even data that was traditionally stored in PCs is migrating to the web (cloud), thereby unlocking the data access from a specific location. Mobile devices complete the anywhere data access vision – they provide access from anywhere. Smart mobile devices combine multiple functions of phones, media players, PCs, etc. Such devices are becoming powerful in their processing power and provide larger storage capacity. These advances provide data access and also enable caching of data (from original sources) that can be processed offline

1.2 Streaming

In certain scenarios, data is simply streamed intelligently through application logic. There is no additional need to store the data – except transiently for the duration of the operation. Conventional database systems require data to be first loaded into the database; then the operation is performed, and the data may be later removed from the database. This adds significant complexity to the application, and dramatically reduces its performance and throughput.

Consider, for example, RFID data in applications. RFID tags are portable sensors that communicate over specialized protocols with RFID reader devices. An increasing number of applications have begun to utilize RFID technology including Manufacturing (e.g. Vendor Managed Inventory, Assembly scheduling), Distribution (e.g. Goods delivery, Shipping verification), Retail (e.g. Shelf stocking, Receiving, Store Replenishment and Ordering, Theft, Merchandise Location), Healthcare Industry to identify patients, Tracking books and automated checkout in libraries, etc. In all these applications, RFID readers (devices) generate events when they identify RFID tags. The RFID event streams are filtered, aggregated, transformed, and correlated so that the events can be monitored in real-time. For example, when a truck carrying packages of product parts (with RFID tags) enters (or leaves) a warehouse, the readers generate events. Spurious events are filtered; related products are aggregated; the event data is transformed and presented on a monitoring dashboard in real-time. The event processing is data-centric and typically requires an in-memory rules engine and query processing.

1.3 Disconnected

Distributed and disconnected application architectures fundamentally change the way applications access and manage data. Instead of locking data in a central location, data is moved closer to the applications, and this enables data processing to be performed in an autonomous and efficient fashion. These applications may run in the mid-tier, on desktops, or on mobile devices. Such an environment is inherently disconnected – there is no need for continuous connectivity between the data sources. Data may be accessed from its original sources, transformed and cached (or stored) close to the applications.

For example, consider a product catalog application aggregating product information across multiple back-end application and data sources. Operations against such an aggregate catalog require them to be decomposed into operations on the underlying sources. After the underlying operations are invoked, responses are collected, and results are aggregated into cohesive responses to the end users and businesses. A typical Catalog Browse operation iterates over a large amount of product data, filters it, personalizes it, and then presents the selected data to the users. These operations are data-centric i.e. they require accessing and querying backend data. Accessing large sets of backend data for every catalog operation can be prohibitively expensive, and can significantly impact the response time and throughput. Caching and querying is therefore a key mechanism in application servers for performance and functionality.

1.4 Embedded

Many applications perform simple to moderate manipulation of data. They need a way of storing, retrieving and manipulating data within the application. These applications themselves are not complex in nature, and are designed to meet a specific user need. Typically, they are developed by vendors specializing in industry verticals (domains) – e.g. Healthcare, Finance etc. These vendors are domain experts but not database specialists. Their focus is the application and would rather not spend their time in database system installation, deployment, and management. While synchronization with backend databases is important, this is typically never seen by the application developer. The databases are typically single application databases; the applications do not want to share (or coordinate) their database with others (which is the case with general purpose database services). These applications could run on devices, desktops, or servers. The ideal way of deployment is deploying the database system components along with the application as a single install. Additionally, different applications may have varying needs from the database system – some may require only ISAM access, while others may require general query support; still others may require synchronization with backend database systems.

While the vertical (embedded) application domain is rapidly growing, traditional DBMS vendors have not paid attention to their needs. Therefore, application vendors have used home grown components for data management using technologies they are familiar with. Files, XML, a rudimentary store and retrieve mechanism, or custom data management implementations are some techniques employed. Consequently, application developers are looking to DBMS vendors for better support for embedded databases to enable their scenarios

It is important to note that new environments and applications span hardware tiers – from devices to desktops to servers and clusters and must work in all tiers.

2 Device Trends

Devices, as relevant to this paper, can be categorized into Mobile devices and Sensor devices. Mobile devices are operated by an end-user to access personal and business data; sensor devices are self-operating and typically collect data from their surrounding environment and provide access to the collected data. Both these devices and their operations impact mobile and embedded database system design and usage. In this section we provide an overview of the advances in these device technologies.

2.1 Mobile Devices

Phones (including smart phones) constitute majority of the mobile devices of interest. A simple phone includes a CPU, memory, storage (flash), networking, and a battery. More recently, phones started offering additional features like camera, music player, GPS, etc. Such convergence of capabilities has lead way to System-on-Chip (SOC) design, with a single chip including a processor, memory, connectivity, camera, music player, video broadcast, GPS, 3D graphics, etc. Smart phones are beginning to run on multi-core processors. Power consumption is still a huge factor in the overall hardware and software system design. Multi core-processors provide asymmetric functionality and can be shutdown and started dynamically as needed to alleviate power consumption. For example, less power consuming operations for multimedia processing can be done on one of the cores. All these advances in mobile phones leads to more data on them, thereby requiring better data management solutions.

2.2 Sensor Devices

Sensor devices are small devices operating with extremely low power consumption (e.g. milliwatts) on batteries. 1000s of these devices are connected, in a geographic network, to form a Sensor Network. Sensor networks are used to collect physical measurements like temperature, humidity, light, etc. in remote environments. A typical sensor device consists of four basic components: sensing, processing, transceiver, and power units. These remote devices are also known as motes. Each mote (e.g. Mica2) contains 4 KB memory, 128 KB – 512 KB flash, with

900MHz multi-channel transceiver. The Mica2 mote weighs about 18 grams and operates on 2 AA batteries lasting for one year.

Each device in the sensor network, known as sensor', is static in one location, gathers data from its surroundings, and streams the data to a central site. Sensors are self-managed (no human interaction) and collect data continuously. They are capable of doing computation and can perform data filtering. Very small footprint database systems (like TinyDB) can run on these mote devices. The federation of databases across the sensor network can send data to a central location in response to a query. In fact, TinyDB supports a variant of SQL as the query language.

2.3 Device Storage Trends

Flash is the primary storage media in both mobile (e.g. phones) and sensor devices. Flash is non-volatile, provides reasonable sizes at affordable prices, and significant advances are taking place in its storage. The capacities are increasing and prices are coming down. The combination of low cost and high capacity are making flash drives very competitive relative to hard disks. In this section we describe some of the key trends in flash storage.

Increase in storage size – Most mobile phones use 1 – 2 GB flash disks today. More applications (like mail, address book, music, pictures) are beginning to take advantage of flash storage. There are also laptop configurations that come with 32 GB flash (currently they are expensive). During the ten years from 1995, flash capacities have doubled every year, growing from 16Mb to 16Gb. By 2012, the capacity is expected to reach 1 Tb. By packaging multiple flash chips, 1 – 2TB disks can be expected in 5 years.

Costs coming down – Today, an 8 Gb (1 GB) flash chip costs about \$5. By 2012, the cost is expected to go down 10 times – a 1 Tb (or 128 GB) is expected to cost about \$50. That is, a 1 TB flash disk will cost about \$400 in 2012. Flash cost is 50X of disks today but expected to be 10X of disks by 2012. Even though Flash is expensive compared to disks, it provides better power utilization and high I/O operations and rates.

Higher IOPS (I/O Operations per Second) – Unlike hard disks, flash storage does not have seek or rotational latencies. Therefore, it can offer higher IOPS for random reads and writes. Current manufacturers claim 10X IOPS compared with hard disks. However, with advances in technology, flash disks with close to 3000 IOPS (for random read/writes) are becoming a reality.

Power consumption – Flash disks consume less power than hard disks, typically in the milliwatt range. Hard disks consume 10 – 15 watts depending on the rotational speed of the disks. Due to the low power consumption, flash is the ideal storage media for sensor devices and mobile phones.

Read/Write characteristics – Flash's read/write mechanism has impact on the database storage design using flash. The read and write operations happen at page granularity. Pages are organized into blocks, typically of 32 or 64 pages. A page can only be written after erasing the entire block to which the page belongs. Page write cost is typically higher than read, and the block erase requirement makes writes even more expensive. A block wears out after 10,000 to 100,000 repeated writes, and so write load should be spread out evenly across the chip. These storage trends warrant some redesign of the storage engine technology in the current mobile and embedded database systems.

These positive trends in device storage are leading way to more data being stored on them that require better data access and management.

3 Mobile and Embedded Applications

In this section, we provide characteristics and examples of mobile and embedded applications.

3.1 Mobile Applications

As employee mobility is increasing, so is the need for traditional desktop applications to run on mobile devices. In the Consumer and Information Worker space, E-mail, Address Book, and Calendaring are three widely used mobile applications. However, these applications are still very simple. As device hardware (processing and memory capacity) advances, users will demand richer capabilities in these applications. For example, consider a rich Calendaring application – with support for checking and scheduling appointments and meetings, sharing calendars across collaborating workers, integrating calendars with other applications, and so on. In the enterprise space, mobile sales personnel will require CRM applications running on their mobile devices; field service employees will need the ability to check product specifications and perform on-line ordering from mobile devices. Following is a list of some representative mobile application scenarios. These are real examples taken from Microsoft's SQL Server Compact Edition customer scenarios, but apply to any mobile DBMS.

Route delivery management – Drivers get route data on a daily basis that is synchronized when they dock their mobile devices. Mobile DMBS provides the local data store on the devices and the data is synchronized to a backend data source

Utilities consumption reading – The solution provides an end to end capability for reading of Oil, Water, Gas and Electricity meters. Field staff use Pocket PC devices to capture meter readings and companies are interested in making the application available through smart phones also.

Mobile CRM – Mobile CRM provides SFA and CRM solution on the devices. The solution typically integrates into other ERP applications. The DBMS provides the local data store and data synchronization (replication) mechanisms. The replication mechanisms work over a variety of transports (e.g. WiFi, Bluetooth, GPRS, 3G, etc).

Sensor Databases – Data collected by the sensor devices is stored in the local DBMS on the device. Such mobile DBMS systems must operate on extremely constrained configurations (e.g. low power, small memory, NVRAM). The sensor devices are typically placed in remote locations (to study remote environments) and monitored from a central site. Such monitoring requires data from individual DBMSs to be queried and aggregated. The network of sensor DBMSs forms a sensor network of federated DBMSs that is queryable from the central site.

3.2 Embedded Applications

The characteristics of embedded applications are described in section 1. Most mobile applications are embedded applications and typically mid-tier applications are embedded and embed a database system (cache) for performance and manageability. Also, most low-end applications are embedded – e.g. Microsoft Access. These applications are self-managed, self-hosted, and very portable. They are developed using simple-to-use developer tools and are also used as offline/local applications. Following are some examples of embedded database applications.

Desktop Media applications – Windows Vista integrates home entertainment into the PC. It delivers an easy and powerful way to manage digital entertainment – photos, music, TV, movies, videos, radio, etc. The SQL CE DBMS is used as an embedded database system for storing this media data e.g. TV listing information is stored and queried using regular SQL; Media Player Playlists and Ratings are stored for efficient organization and query; Photo Organization data is stored for flexible organization and ad-hoc query.

Line of Business applications – Typical LOB applications are multi-tier applications where data in the back-end data source tends to be authoritative. Data is cached in the middle-tier as reference data and application logic executes over it. This reference data is typically integrated from multiple backend data/application sources, transformed into a format suitable for application logic to process efficiently, and brought close to the application in the mid-tier. Also, the reference data is often read-only and suitable for caching within the application. Embedded Database systems are used for such reference data caching and they provide rich storage management and query on the cached data. As explained earlier, embedded DBMSs also make application packaging and installation easier. For similar reasons, embedded database systems are also used as local caches or stores in client applications. Another important scenario in LOB applications is offline experience. Consider a sales person who routinely maintains her customer relationships. The sales person, even while not connected to the corporate network, must be able to create new customers, update customer information, and gather requirements. This is an example of the mobile CRM scenario which requires the application to be executed while the salesperson is mobile and offline. It should also be noted that even web application architectures follow similar application and data processing patterns, requiring the need for embedded database systems in the web application tier and even in the browser. For example, recently Google released a browser extension (Google Gears) that embeds SQLite to provide offline capabilities to browser applications.

Stream processing – In section 1, we described the streaming data environment. Stream processing is different from data processing of traditional relational database systems. In stream processing engines, data is processed as it arrives and before it is stored. In-memory embedded DBMS systems can be used in such stream processing engines.

4 Mobile and Embedded DBMS Characteristics

The data access and management requirements of the applications described above are significantly different from that of traditional server DBMSs. These new applications must be able to run on multiple tiers ranging from devices to servers to web and would benefit from various existing database mechanisms. However, these database mechanisms (like query, indexing, persistence) must be unlocked from the traditional monolithic DBMSs and made available as embeddable components (e.g. DLLs) that can be embedded within applications, thereby, enabling them to meet the requirements described above. Such Mobile and Embedded DBMSs have the following characteristics:

1. **Embeddable in applications** – Mobile and Embedded DBMSs form an integral part of the application or the application infrastructure, often requiring no administration. Database functionality is delivered as part of the application (or app infrastructure). While the database must be embeddable as a DLL in applications, it must also be possible to deploy it as a stand-alone DBMS with support for multiple transactions and applications.
2. **Small footprint** – For many applications, especially those that are downloadable, it is important to minimize DBMS footprint. Since the database system is part of the application, the size of the DBMS affects the overall application footprint. In addition to the small footprint, it is also desirable to have short code paths for efficient application execution. Most of these applications do not require the full functionality of commercial DBMSs; they require simple query and execute in constrained environments.
3. **Run on mobile devices** – The DBMSs that run on mobile devices tend to be specialized versions of mobile and embedded DBMSs. In addition to handling the memory, disk and processor limitations of these devices, the DBMS must also run on specialized operating systems. The DBMS must be able to store and forward data to the back-end databases as synchronization with backend systems is critical for them.
4. **Componentized DBMS** – Often, to support the small footprint requirement, it is important to include only the functionality that is required by the applications. For example, many simple applications just require ISAM like record-oriented access. For these applications, there is no need to include the query processor, thereby increasing the footprint. Similarly, many mobile and mid-tier applications require only a small set of relational operators while others require XML access and not relational access. So, it should be possible to pick and choose the desired components.
5. **Self managed DBMS** – The embedded DBMS is invisible to the application user. There can be no DBA to manage the database and operations like backups, recovery, indexing, tuning etc. cannot be initiated by a DBA. If the database crashes, the recovery must start instantaneously. The database must be self managed or managed by the application. Also, embedded DBMS must auto install with the application – it should not be installed explicitly (user action) or independently. Similarly when the application is shut down, the DBMS must transparently shutdown.

6. **In-Memory DBMS** – These are specialized DBMSs serving applications that require high performance on data that is small enough to be contained in main memory. In-memory DBMSs require specialized query processing and indexing techniques that are optimized for main memory usage. Such DBMSs also can support data that may never get persisted.
7. **Portable databases** – There are many applications which require very simple deployment – installing the application should install the database associated with it. This requires the database to be highly portable. Typically, single file databases (e.g. like Microsoft Access databases) are ideally suited for this purpose. Again, there should be no need to install the DBMS separately – installing the application installs the DBMS and then copying the database file completes the application migration.
8. **No code in the database** – Portable database must also be safe. Executable code can be a carrier of virus or other threats. By eliminating any code storage in the database, it can be made safer and portable.
9. **Synchronize with back-end data sources** – In the case of mobile and cached scenarios, it must be possible to synchronize the data with the back-end data sources. In typical mid-tier (application server) caches, the data is fetched from the back-end databases into the cache, operated on, and synchronized with the back-end database.
10. **Remote management** – While mobile and embedded DBMSs must be self managed, it is important to allow them to be managed remotely also, especially those on mobile devices. In enterprises (e.g. FedEX, UPS), mobile devices must be configured and managed in a manner compliant with the company standards. Therefore centralized remote management of these devices is necessary.
11. **Custom programming interfaces** – An important usage of embedded DBMS is in specialized data-centric applications. Such applications use variety of data models (e.g. Relational, XML, Streams, and Analytics) and query languages. The embedded DBMSs must be componentized and extensible to allow (application) domain-specific query languages and programming surfaces.

4.1 Mobile vs. Embedded DBMS

While both mobile and embedded DBMSs share many common characteristics, there are also differences that separate them, especially in deployment. In fact, mobile DBMSs are typically embedded DBMSs but considerably constrained by the environment in which they must execute and perform. The following table illustrates key differences between the two:

Mobile DBMS	Embedded DBMS
Targets device tier. Supports device scenarios	Targets all tiers. Deployment is application-specific
Constrained by device physical characteristics	Constrained by the deployment environment
Power, memory size impact the design	Power and media are not constraints
Size (small footprint) is critical	Small size is important
Componentization is not critical but helps minimize size	Componentization is critical to support varied deployments
Scale and throughput are not critical	Scale and throughput are important

5 Mobile and Embedded DBMS Design Considerations

While the architecture of mobile and embedded DBMSs is similar to that of traditional relational DBMSs, the characteristics described in the previous section must be factored in. Some of these characteristics are more critical than others – componentization, small footprint, and self-management are by far the most critical characteristics. In fact, good componentized DBMS architecture naturally makes way for other characteristics like embeddability, size, deployment, etc. This section describes the key design considerations for addressing the mobile and embedded DBMS requirements.

5.1 Componentization

The components of a mobile and embedded DBMS are not really new but how well the functionality is factored and layered within and across the components is important. The key high level components are: Storage Engine, Query Processor, Data Access APIs, and Synchronization. Since specialized database systems and embedded applications know the specific database functionality they desire, it must be possible to choose the specific components and functionality. For these applications one size does not fit all. Custom application-specific packaging offers both performance and size benefits. If an embedded application is single-threaded and does not require isolation, lock management can be factored out. Similarly, if an application does not require JOINs, the join code can be factored out from the query processor – thereby reducing the overall size.

Componentization also provides extensibility. For example, consider processing of structured (Relational) and semi-structured (XML) data with SQL and XQuery respectively. In implementing DBMSs for such support, we believe a common storage engine component can be used with two query processing components, one for SQL and the other for XQuery. Such an architecture not only allows for reuse of components, it also allows for extensibility for plugging additional application domain-specific query processors.

Architecturally, a well factored DBMSs forms an inverted triangle of components, with one (or very small number of) storage engines at the bottom and multiple query execution engines, query optimizers, query compilers, APIs layers, and language bindings at the top.

In the next few sections we look at sub-componentization within these higher level components.

5.2 Storage Engine

Most storage engines support media management to record/row management with transactions, recovery, and efficiency. The storage engine can be componentized as follows:

- Media management (disk, flash, and memory) – While most mobile and embedded DBMS's storage engines must support data on disks, they are also embedded in applications whose data is primarily memory-resident. The storage engine must turn off persistence to disk and optimize large memory use. Mobile DBMS storage engines must support flash media, when they are used in mobile and sensor devices. Supporting different media requires different storage and access mechanisms. The storage engine must be extensible to support these mechanisms.
- Transactions – Embedded DBMS must be capable of supporting concurrency control (e.g. locking) and transactions. However, not all embedded applications require the full ACID properties. Most applications require atomicity but the other properties can be optional. For example, a single threaded application does not require isolation; a pure main-memory based application does not require persistence (and durability). Most read-only caches do not even require atomicity; that is, transaction support is optional. In such cases, the storage engine functionality must be componentized so that these applications that don't require transactions can either bypass the transaction management or factor it out of their application. Also, when embedded DBMSs are used as application caches, where the authoritative data comes from backend data sources, data versioning and multi-versioned concurrency control can improve the overall cache performance. However, adding any mechanism can increase the overall footprint (size) but proper componentization allows the application to choose the desired components.
- Access methods – Since embedded DBMSs are used in variety of application scenarios in different storage environments (e.g. with large memory, flash, disk), the storage and access methods must be optimized to take advantage of this environment. For example, in in-memory BI systems, column based storage and compression is desirable for high performance and memory utilization. Similarly hash based access methods are more appropriate for key based access in large memory environments. Broadly, embedded DBMSs must support flexible row formats and multiple access methods (B-Tree, Hash, and Heap).

5.3 Query Processor

The query language requirements tend to be driven by the embedded applications. LOB applications typically require SQL (or some variation) support; document applications may require XPath or XQuery; analytics applications require multi-dimension query; streaming applications may require some form of stream SQL; and so on. Since there is no single query language that can support all embedded applications, either query processors should be optional in the embedded DBMSs or their architecture should be extensible so that multiple query processors can be plugged-in. The query processor itself can be componentized as follows:

- Small number of query execution operators – for example, support for extended relational, analytics operators. The query execution operator interfaces must be exposed so that multiple query languages can be supported on them.
- Multiple query compilers – typical query compiler includes the language processor and the optimizer. These query compilers optimize the queries and generate an execution plan in terms of the query execution operators described above. Again, such an architecture allows for reuse (of execution operators) and extensibility.
- Both the query execution and the query compiler components must be configurable so that applications can pick and choose the desired query functionality.

5.4 Synchronization

Synchronization mechanisms replicate or synchronize data between the application and the backend data sources. Synchronization is important for mobile and embedded applications for the following reasons:

- Mobile and embedded DBMSs are predominantly deployed in the client-tier and mid-tier.
- Data is copied or replicated from the backend data sources close to the applications (e.g. as caches) in the client and middle tier, for better performance and ease of application management.
- Occasional disconnection between the application and the backend data sources causes data to be off-lined into the client databases. Offline databases must be periodically synchronized with the original backend data sources.
- Synchronization must work over variety of communication networks (e.g. LAN, WAN, wireless networks, etc), and to variety of data sources (e.g. database systems, application systems, web services).

5.5 Management

For most mobile and embedded applications, database management is transparent. That is the embedded DBMS is self-managed. In traditional DMBSs, DBAs perform the following management functions but in embedded DBMSs these are done transparently:

- Backup and restore – Unlike in traditional DBMSs, in mobile and embedded DBMSs the backup operation is very simple and is done transparently. Applications specify a policy (typically, through an API) that will schedule periodic but consistent copying of the database files. It is highly desirable to have a single file database so that consistent copying becomes easier. Some embedded DBMSs do not even require a separate log. Restore is essentially copying the consistent copy of the file to the desired location. In systems which have logs, it is necessary to copy the log, along with the database, in a consistent manner without the application being aware of the different files.
- Recovery – Again, the application is not aware of recovery. It is done transparently and immediately as soon as failure is detected. For example, if a transaction is aborted, undo is done transparently. If the system fails, then on restart, the database must be recovered automatically. In log-less systems (i.e. systems where there is no in-place writing), this operation is trivial. In log based systems, the log must be applied transparently.

- Table and index reorganizations – In most mobile and embedded applications, the database workload (query and updates) are known at application development time. Therefore, table and index creations are typically done explicitly. These applications are well tuned before they are deployed. Therefore, dynamic index creation is not a requirement. However, table and index storage re-organization (e.g. compaction) is done automatically, typically at the database level to minimize fragmentation.

6 Conclusions

With the increased interest in specialized applications and database systems (like streaming, analytics, mobile, sensor networks, etc.), the need for Mobile and Embedded DBMSs is increasing. There have been mobile and embedded DBMS products in the market that satisfy some of the characteristics described above, but were designed as low-end database products and do not factor in the recent hardware and application trends. For example, products that focused on small footprint but did not design with componentization for embedded applications; similarly, there are products that only provide embedded (well componentized) storage engines but without any query processing components.

The hardware trends in multi core CPUs, large main memory, and NVRAMs (e.g. flash) are beginning to disrupt the traditional database architectures. Particularly, advances in large memory and flash storage open doors for further research in database architectures. The componentized mobile and embedded DBMSs are in a position to adapt to these changing hardware and software trends more rapidly.

References

[1] *http://research.microsoft.com/G̃ray/talks/Flash_is_Good.ppt.*

[2] Michael Stonebraker and Ugur Cetintemel. *One Size Fits All: An Idea Whose Time Has Come and Gone*.

[3] Michael Olson, Keith Bostic, Margo Seltzer. Berkeley DB. *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*.

[4] Margo Seltzer. There is more to data access than SQL. In *ACM Queue, Databases, Vol. 3 No. 3 - April 2005*.

[5] *SQL Server CE: http://www.microsoft.com/sql/editions/compact/default.mspx*.

[6] Suman Nath and Aman Kansai. Dynamic Self-tunig Database for NAND Flash. *ISPN '07, April 25 - 27, 2007, Cambridge, Massachusetts, USA*.

[7] *TinyDB: http://telegraph.cs.berkeley.edu/tinydb/*.

[8] *Window CE Platform, Microsoft Corporation*.

Caching and Replication in Mobile Data Management

Evaggelia Pitoura
Computer Science Department,
University of Ioannina, Greece
pitoura@cs.uoi.gr

Panos K. Chrysanthis
Department of Computer Science,
University of Pittsburgh, USA
panos@cs.pitt.edu

Abstract

Mobile data management has been an active area of research for the past fifteen years. Besides dealing with mobility itself, issues central in data management for mobile computing include low bandwidth, intermittent network connectivity and scarcity of resources with emphasis on power management. In this article, we focus on how caching and replication in mobile data management address these challenges. We consider two antagonistic criteria, that of ensuring quality of data in terms of consistency and coherency and that of achieving quality of service in terms of response time and availability.

1 Introduction

Mobile computing refers to computing using devices that are not attached to a specific location, but instead their position (network or geographical) changes. Mobile computing can be traced back to file systems and the need for disconnected operations in the end of the 80s. With the rapid growth in mobile technologies and the cost effectiveness in deploying wireless networks in the 90s, the goal of mobile computing was to support of AAA (anytime, anywhere and any-form) access to data by users from their portable computers and mobile phones, devices with small displays and limited resources. This led to research in mobile data management including transaction processing, query processing and data dissemination [22]. A key characteristic of all these research efforts was the great emphasis on the mobile computing challenges, including:

- *Intermittent Connectivity* This refers to the fact that computation must proceed despite short or long periods of network unavailability.
- *Scarcity of Resources* Due to the small sizes of portable devices, there are implicit restrictions in the availability of storage and computation capacity and mostly of energy.
- *Mobility* The implications of mobility are varying. First, mobility introduces a number of technical challenges at the networking layer. It also offers a number of opportunities at the higher layers for explicitly using location information either at the semantic level (for instance, for providing personalization) or at the performance level (for instance, for data prefetching). Finally, it amplifies heterogeneity.

In general, one can distinguish between single-hop and multi-hop underlying infrastructures. In *single-hop* infrastructures, each mobile device communicates with a stationary host, which corresponds to its point

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

of attachment to the network. The rest of the routing is the responsibility of a stationary infrastructure, e.g., the Internet. On the other hand, in *multi-hop* wireless communication, an ad-hoc wireless network is formed in which, wireless hosts participate in routing messages among each other. In both infrastructures, the hosts between the source (or sources) that holds the data and the requester of data (or data sink) form a dissemination tree. The hosts (mobile or stationary) that form the dissemination tree may store data and take part in the computation towards achieving *in network* processing (e.g., aggregation). Locally caching or replicating data at the wireless host or at the intermediate nodes of the dissemination tree are important for improving system performance and availability.

Caching and replication generally attempt to guarantee that most data requests are for data that is being held in main memory or local storage, negating the need to perform I/O, or a remote data retrieval. Hence, the use of appropriate caching/replication schemes have been traditionally used to improve performance and reduce service time. In mobile environments the performance considerations go beyond simple speedups and data retrieval delays. In this article, we examine how caching and replication has been utilized in mobile data management and more specifically in the first infrastructure where data are cached at the mobile device in order to avoid excessive energy consumption and to cope with intermittent connectivity.

In this paper, our focus is on consistency, that is, how to ensure the correctness of operations on cached data. In Section 2, we provide a brief taxonomy of related correctness criteria. While traditionally, cached data are read-only, in mobile computing, some restricted form of cache updates is supported especially in case of disconnections. We call this form of caching that allows cache updates at the client, *two-tier caching* and discuss it in detail in Section 3. In Section 4, we present issues related to disseminating updates from the rest of the network to the mobile device. Finally, Section 5 concludes the paper.

2 Consistency Levels

We consider the case in which a mobile computing device (such as a portable computer or cellular phone) is connected to the rest of the network typically through a wireless link. Wireless communication has a double impact on the mobile device since the limited bandwidth of wireless links increases the response times for accessing remote data from a mobile host and transmitting as well as receiving of data are high energy consumption operations.

The principal goal is to store appropriate pieces of data locally at the mobile device so that it can operate on its own data, thus reducing the need for communication that consumes both energy and bandwidth. At some point, operations performed at the mobile device must be synchronized with operations performed at other sites. The complexity of this synchronization depends greatly on whether updates are allowed at the mobile device. The main reason for allowing such updates is to sustain network disconnections. When there are no local updates, the important issue is disseminating updates from the rest of the network to the mobile device.

Synchronization depends on the level at which correctness is sought. This can be roughly categorized as replica-level correctness and transaction-level correctness. At the *replica level*, correctness or coherency requirements are expressed per item in terms of the allowable divergence among the values of the copies of each item. There are many ways to characterize the divergence among copies of an item. For example, with *quasi copies* [3], the coherency or freshness requirements between a cached copy of an item and its primary at the server are specified by limiting (a) the number of updates (versions) between them, (b) their distance in time, or (c) the difference between their values. At the *transaction level*, the strictest form of correctness is achieved through global serializability that requires the execution of all transactions running at mobile and stationary hosts to be equivalent to some serial execution of the same transactions. In case of replication, one copy serializability provides equivalence with a serial execution on a one-copy database. One-copy serializability does not allow any divergence among copies.

There is a large number of correctness criteria proposed besides serializability. A practical such criterion is snapshot isolation [5]. With snapshot isolation, a transaction is allowed to read data from any database snapshot at a time earlier than its start time. Central are also criteria that treat read-only transactions, i.e., transactions with no update operations, differently. Consistency of read-only transactions is achieved by ensuring that transactions read a database instance that does not violate any integrity constraints (as for example with snapshot isolation), while freshness of read-only transactions refers to the freshness of the values read [11]. Finally, *relaxed-currency serializability* allows update transactions to read out-of-date values as long as they satisfy some freshness constraints specified by the users [6].

There are two basic ways of propagating updates. Eager replication synchronizes all copies of an item within a single transaction, whereas with lazy replication, transactions for keeping replica coherent run as separate, independent database transactions after the original transaction. One-copy serializability as well as other forms of correctness can be achieved either through eager or lazy update propagation.

3 Two-tier Caching

In this section, we assume that data can be updated at the mobile device. The main motivation is support for disconnected operation. *Disconnected operation* refers to the autonomous operation of a mobile client, when network connectivity becomes unavailable for instance, due to physical constraints, or undesirable, for example, for reducing power consumption. Preloading or prefetching data to sustain a forthcoming disconnection is often termed *hoarding*. The content of data to be prefetched may be determined (a) automatically by the system by utilizing implicit information, most often based on the past history of data references, or (b) by instructions given explicitly by the users, as in *profile-driven data prefetching* [7], where a simple profile language is introduced for specifying the items to be prefetched along with their relative importance. Additional information such as a set of allowable operations or a characterization of the required data quality may also be cached along with data. For example, in the *Pro-Motion infrastructure* [28, 15], the unit of caching and replication is a *compact*, an object that encapsulates the cached data, operations for accessing the cached data, state information (such as the number of accesses to the object), consistency rules that must be followed to guarantee consistency, and obligations (such as deadlines).

To allow concurrent operation at both the mobile client and other sites during disconnection, optimistic approaches to consistency control are typically deployed. Optimistic consistency maintenance protocols allow data to be accessed concurrently at multiple sites without a priori synchronization between the sites, potentially resulting in short term inconsistencies. Such protocols trade-off quality of data for improving quality of service. Optimistic replication has been extensively studied as a means for consistency maintenance in distributed systems (for example, see [23] for a thorough recent survey). In this paper, we present some representative examples of optimistic protocols in the context of mobile computing.

Consistent operation during disconnected operation has been also extensively addressed in the context of *network partitioning*. In this context, a network failure partitions the sites of a distributed database system into disconnected clusters. Various approaches have been proposed and are excellently reviewed in [8]. In general, protocols in network partition follow peer-to-peer models where transactions executed in any partition are of equal importance, whereas the related protocols in mobile computing most often consider transactions at the mobile host as second-class, for instance, by considering their updates as tentative. Furthermore, disconnections in mobile computing are common and some of them may be considered foreseeable.

Disconnections correspond to the extreme case of total lack of connectivity. Other connectivity constraints, such as weak or intermittent connectivity also affect the protocols for enforcing consistency. In general, weak connectivity is handled by appropriately revising those operations whose deployment involves the network. For instance, the frequency of propagation to the server of updates performed at the local data may depend on connectivity conditions.

In early research in mobile computing, a general concern has been whether issues such connectivity or mobility should be transparent or hidden from the users. In this respect, adapting the levels of transaction or replica correctness to the system conditions such as the availability of connectivity or the quality of the network connection and providing applications with less than strict notions of correctness can be seen as making such conditions visible to the users. This is also achieved by explicitly extending queries with quality of data specifications for example for constraining the divergence between copies.

Some common characteristics of protocols for consistency in two-tier caching are:

- the propagation of updates performed at the mobile site follow in general lazy protocols,
- reads are allowed at the local data, while updates of local data are tentative in the sense that they need to be further validated before commitment.
- for integrating operations at the mobile hosts with transactions at other sites, in the case of replica-level consistency, copies of an item are reconciled following some conflict resolution protocol. At the transaction-level, local transactions are validated against some application or system level criterion. If the criterion is met, the transaction is committed. Otherwise, the execution of the transaction is either aborted, reconciled or compensated. Such actions may have cascaded effects on other tentatively committed transactions that have seen the results of the transaction.

Next, we present a number of consistency protocols that have been proposed for mobile computing.

Isolation-Only transactions in Coda Coda [24] is one of the first file systems designed to support disconnections and weak connectivity. Coda introduced *isolation-only* transactions (IOTs) [14] in file systems. An IOT is a sequence of file access operations. A transaction T is called a *first-class transaction*, if it does not have any partitioned file access, i.e., the mobile host maintains a connection for every file it has accessed. Otherwise, T is called a *second-class transaction*. Whereas the result of a first-class transaction is immediately committed, a second-class transaction remains in the pending state till connectivity is restored. The result of a second-class transaction is held within the local cache and visible only to subsequent accesses on the same host. Second-class transactions are guaranteed to be locally serializable among themselves. A first-class transaction is guaranteed to be serializable with all transactions that were previously resolved or committed at the fixed host. Upon reconnection, a second-class transaction T is validated against one of the following two serialization constraints. The first is global serializability, which means that if a pending transaction's local result were written to the fixed host as is, it would be serializable with all previously committed or resolved transactions. The second is a stronger consistency criterion called global certifiability (GC) which requires a pending transaction be globally serializable not only with, but also after all previously committed or resolved transactions.

Two-tier Replication With *two-tier replication* [12], replicated data have two versions at mobile nodes: master and tentative versions. A master version records the most recent value received while the site was connected. A tentative version records local updates. There are two types of transactions analogous to second- and first-class IOTs: tentative and base transactions. A *tentative transaction* works on local tentative data and produces tentative data. A *base transaction* works only on master data and produce master data. Base transactions involve only connected sites. Upon reconnection, tentative transactions are reprocessed as base transactions. If they fail to meet some application-specific acceptance criteria, they are aborted.

Two-Layer Transactions With two-layer transactions [17], transactions that run solely at the mobile host are called *weak*, while the rest are called *strict*. A distinction is drawn between weak copies and strict copies. In contrast to strict copies, weak copies are only tentatively committed and hold possibly obsolete values. Weak transactions update weak copies, while strict transactions access strict copies located at any site. Weak copies are integrated with strict copies either when connectivity improves or when an application-defined freshness limit to the allowable deviation among weak and strict copies is passed. Before reconciliation, the results of weak transactions are visible only to weak transactions at the same site. Strict transactions are slower than weak transactions, since they involve the wireless link but guarantee permanence of updates and currency of reads. During disconnection, applications can only use weak transactions. In this case, weak transactions have similar semantics with second-class IOTs [14] and tentative transactions [12]. Adaptability is achieved by restricting the number of strict transactions depending on the available connectivity and by adjusting the application-defined degree of divergence among copies.

Bayou Bayou [27, 26, 16] is built on a peer-to-peer architecture with a number of replicated servers weakly connected to each other. Bayou does not support full-fledged transactions. A user application can read-any and write-any available copy. Writes are propagated to other servers during pair-wise contracts called *anti-entropy* sessions. When a write is accepted by a Bayou server, it is initially deemed tentative. As in two-tier replication [12], each server maintains two views of the database: a copy that only reflects committed data and another full copy that also reflects the tentative writes currently known to the server. Eventually, each write is committed using a primary-commit schema. That is, one server designated as the primary takes responsibility for committing updates. Because servers may receive writes from clients and other servers in different orders, servers may need to undo the effects of some previous tentative execution of a write operation and re-apply it. The Bayou system provides dependency checks for automatic conflict detection and merge procedures for resolution. Instead of transactions, Bayou supports *sessions*. A session is an abstraction for a sequence of read and write operations performed during the execution of an application. *Session guarantees* are enforced to avoid inconsistencies when accessing copies at different servers; for example, a session guarantee may be that read operations reflect previous writes or that writes are propagated after writes that logically precede them. Different degrees of connectivity are supported by individually selectable session guarantees, choices of committed or tentative data, and by placing an age parameter on reads. Arbitrary disconnections among Bayou's servers are also supported since Bayou relies only on pair-wise communication. Thus, groups of servers may be disconnected from the rest of the system yet remain connected to each other.

4 Update Dissemination

In this section, we consider data at the mobile device to be read-only, as in traditional client-server caching [9]. In this case, the main issue is developing efficient protocols for disseminating server updates to mobile clients. Most such cache invalidation protocols developed for mobile computing focus on the case in which a large number of clients is attached to a single server. Often, the server is equipped with an efficient broadcast facility that allows it to propagate updates to all of its clients. Different assumptions are made on whether the server maintains or not any information about which clients it is serving, what are the contents of their cache, and when their cache was last validated. Servers that hold such information are called *stateful*, while servers that do not are called *stateless*. Another issue pertinent to mobile computing is again handling disconnections, in particular, ensuring that cache invalidation are received by clients despite any temporary network unavailability.

Update propagation may be either synchronous or asynchronous. In *synchronous* methods, the server broadcasts an invalidation report periodically. A client must listen for the report first to decide whether its cache is valid or not. Thus, each client is confident for the validity of its cache only as of the last invalidation report. This adds some latency to query processing, since to answer a query, a client has to wait for the next invalidation report. In case of disconnections, synchronous methods surpass asynchronous in that clients need only periodically tune in to read the invalidation report instead of continuously listening to the broadcast. However, if the client remains inactive longer than the period of the broadcast, the entire cache must be discarded, unless special checking is deployed.

Invalidation protocols vary in the type of information they convey to the clients. In case of replica level correctness, it suffices that single read operations access current data. In this case, invalidation may include just a list of the updated items or in addition to this, their updated values. Including the updated values may be wasteful of bandwidth especially when the corresponding items are cached at only a few clients. On the other hand, if the values are not included, the client must either discard the item from its cache or communicate with the server to receive the updated value. The reports can provide information for individual items or aggregate information for sets of items. In case of transaction level correctness, invalidation reports must include additional information regarding server transactions.

The efficiency of an update dissemination protocol depends on the connectivity behavior of the mobile clients. In [4], clients that are often connected are called workaholic, while clients that are often disconnected are the sleepers. Three synchronous strategies for stateless servers are considered. In the broadcasting *timestamps* strategy (TS), the invalidation report contains the timestamps of the latest change for items that have had updates in the last w seconds. In the *amnestic terminals* strategy (AT), the server only broadcasts the identifiers of the items that changed since the last invalidation report. In the *signatures* strategy, signatures are broadcast. A signature is a checksum computed over the value of a number of items by applying data compression techniques similar to those used for file comparison. Each of these strategies is shown to be effective for different types of clients. Signatures are best for long sleepers, that is, when the period of disconnection is long and hard to predict. The AT method is best for workaholic. Finally, TS is shown to be advantageous when the rate of queries is greater than the rate of updates provided that the clients are not workaholics.

Another model of operation in the content of mobile databases is that of a broadcast or push model [1]. In this model, the server broadcasts (periodically) data to a set of mobile clients. Clients monitor the broadcast and retrieve the data items they need as they arrive. Data of interest may also be cached locally at the client.

When clients read data from the broadcast, a number of different replica-level correctness models are reasonable [2]. For example, if clients do not cache data, the server always broadcasts the most recent values, and there is no backchannel for on-demand data delivery, then the *latest value* model is a model that arise naturally. In this model, clients read the most recent value of a data item. Methods for enforcing transaction-level correctness are presented in [19]. With the *invalidation* method, the server broadcasts an invalidation report with the data items that have been updated since the broadcast of the previous report. Transactions that read obsolete items are aborted. With the *serialization graph testing* (SGT) method, the server broadcasts control information related to conflicting operations. Clients use this information to ensure that their read-only transactions are serializable with the server transactions. With *multiversion broadcast* [20, 18], multiple versions of each item are broadcast, so that client transactions always read a consistent database snapshot.

A general theory of correctness for broadcast databases as well as the fundamental properties underlying the techniques for enforcing it are given in [21]. Correctness characterizes the freshness of the values seen by the clients with regards to the values at the server as well as the temporal discrepancy among the values read by the same transaction.

More recently, the concept of materialized views was extended in the context of mobile databases to operate in a fashion similar to data caches supporting local query processing [13]. As in traditional databases, materialized views in mobile databases provide a means to present different portions of the databases based on users' perspectives and, similar to data warehouses, materialized views provide a mean to support personalized information gathering from multiple data sources. Personalization is expressed in the form of view maintenance options for recomputation and incremental maintenance. They offer a finer grain of control and balance between data availability and currency, the amount of wireless communication and the cost of maintaining consistency. In order to better characterize these personalizations, in [13] *recomputational consistency* was introduced and the *materialized view consistency* [30] was enhanced with new levels which correspond to specific view currency customizations.

5 Summary

In this short article, we presented issues related to cache consistency in mobile computing. The methods presented can be considered as extensions of traditional client-server caching, where the client is a mobile device.

The main motivation for this form of caching is improving availability especially in the case of network disconnections. Caching also improves performance through reducing the communication overhead in terms of both data access delays and energy consumption.

An interesting extension of the current methods is hierarchical caching for the emerging infrastructures of multi-hop wireless networks. Besides the challenges due to mobility, hierarchical caching introduces new complications such as the multiple levels of intervening caches can that create adverse workloads for the caching schemes used at different levels. A hierarchical caching scheme must have the ability to adapt itself, thereby acting synergistically and cooperatively with other caching schemes on mobile peers [10].

Finally, mobile computing is often related to wireless computing and to computation involving small devices, including sensors or RFID tags. In the case of sensors, their limited power restricts the amount of processing and communication that sensors can perform before they become inactive. Thereby, an interesting question related to hierarchical caching is how caching at different sensors can help in the conservation of their energy, thereby prolonging the lifetime of a sensor network and improving the quality of the data [25, 29].

References

[1] S. Acharya, M. J. Franklin, and A. B. Zdonik. Balancing push and pull for data broadcast. In *SIGMOD Conference*, pages 183–194, 1997.

[2] S. Acharya, M. J. Franklin, and S. B. Zdonik. Disseminating updates on broadcast disks. In *VLDB*, pages 354–365, 1996.

[3] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems (TODS)*, 15(3):359–384, September 1990.

[4] D. Barbará and T. Imielinski. Sleepers and workaholics: Caching strategies in mobile environments. *VLDB J.*, 4(4):567–602, 1995.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil., and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the ACM SIGMOD Conference*, pages 1–10, 1995.

[6] P. A. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma. Relaxed-currency serializability for middle-tier caching and replication. In *SIGMOD Conference*, pages 599–610, 2006.

[7] M. Cherniack, E. F. Galvez, M. J. Franklin, and S. B. Zdonik. Profile-Driven Cache Management. In *Proceedings of the ICDE Conference*, pages 645–656, 2003.

[8] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in Partitioned Networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[9] M. J. Franklin, M. J. Carey, and M. Livny. Transactional client-server cache consistency: Alternatives and performance. *ACM Trans. Database Syst.*, 22(3):315–363, 1997.

[10] P. K. Chrysanthis G. Santhanakrishnan, A. Amer. Towards universal mobile caching. In *Proc. of the 4th ACM Int'l Workshop on Data Engineering for Wireless and Mobile Access*, June 2005.

[11] H. Garcia-Molina and G. Wiederhold. Read-Only Transactions in a Distributed Database. *ACM Transactions on Database Systems*, 7(2):209–234, June 1982.

[12] J. Gray, P. Helland, P. O' Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proceedings of the ACM SIGMOD Conference*, pages 173–182, Montreal, Canada, 1996.

[13] S. Weissman Lauzac and P. K. Chrysanthis. Personalizing Information Gathering for Mobile Database Clients. In *Proceedings of the ACM Annual Symposium on Applied Computing*, pages 49–56, 2002.

[14] Q. Lu and M. Satyanarayanan. Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, Washington, May 1995.

[15] S. Mazumdar and P. K. Chrysanthis. Localization of Integrity Constraints in Mobile Databases and Specification in PRO-MOTION. *ACM Mobile Networks*, 9(5):481–490, October 2004.

[16] K. Petersen, M. Spreitzer, D. B. Terry, M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, pages 288–301, 1997.

[17] E. Pitoura and B. Bhargava. Data Consistency in Intermittently Connected Distributed Systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(6):896–915, November 1999.

[18] E. Pitoura and P. K. Chrysanthis. Exploiting versions for handling updates in broadcast disks. In *VLDB*, pages 114–125, 1999.

[19] E. Pitoura and P. K. Chrysanthis. Scalable processing of read-only transactions in broadcast push. In *ICDCS*, pages 432–439, 1999.

[20] E. Pitoura and P. K. Chrysanthis. Multiversion data broadcast. *IEEE Trans. Computers*, 51(10):1224–1230, 2002.

[21] E. Pitoura, P. K. Chrysanthis, and K. Ramamritham. Characterizing the temporal and semantic coherency of broadcast-based data dissemination. In *ICDT*, pages 410–424, 2003.

[22] E. Pitoura and G. Samaras. *Data Management for Mobile Computing*. Kluwer Academic Publishers, 1998.

[23] Y. Saito and M. Shapiro. Optimistic Replication. *ACM Computing Surveys*, 37(1):42–81, March 2005.

[24] M. Satyanarayanan. The evolution of coda. *ACM Trans. Comput. Syst.*, 20(2):85–124, 2002.

[25] M. A. Sharaf, J. Beaver, A. Labrinidis, and P. K. Chrysanthis. Balancing energy efficiency and quality of aggregate data in sensor networks. pages 13(4): 384 – 403, December 2004.

[26] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, pages 140–149, September 1994.

[27] D. B. Terry, M. M Theimer, K. Petersen, A. J. Demers, M. J Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[28] G. Walborn and P. K. Chrysanthis. PRO-MOTION: Support for Mobile Database Access. *Personal and Ubiquitous Computing*, 1(3), September 1997.

[29] D. Zeinalipour-Yazti, P. Andreou, P. K. Chrysanthis, and G. Samaras. Mint views: Materialized in-network top-k views in sensor networks. In *Proceedings of the 7th International Conference in Mobile Data Management*, pages 182–189, May 2007.

[30] Y. Zhuge, H. Garcia-Molina, and J. Wiene. Consistency Algorithms for Multi-Source Warehouse View Maintenance. *Distributed and Parallel Databases*, 4(4), 1997.

Berkeley DB: A Retrospective

Margo Seltzer, Oracle Corporation

Abstract

Berkeley DB is an open-source, embedded transactional data management system that has been in wide deployment since 1992. In those fifteen years, it has grown from a simple, non-transactional key-data store to a highly reliable, scalable, flexible data management solution. We trace the history of Berkeley DB and discuss how a small library provides data management solutions appropriate to disparate environments ranging form cell phones to servers.

1 Introduction

Berkeley DB is an open-source, embedded transactional data management system that has been in wide deployment since 1992. In this context, embedded means that it is a library that is linked with an application, providing data management completely hidden from the end-user. Unlike conventional database management solutions, Berkeley DB does not have a separate server process and requires no manual database administration. All administration is built directly into the applications that embed it.

As Berkeley DB is an embedded data management system, claims of performance and scale are only meaningful in the context of applications that use it. Berkeley DB provides data management for mission-critical web applications and web sites. For example, Google Accounts uses Berkeley DB to store all user and service account information and preferences [12]. If this application/database goes down, users cannot access their email and other online services; reliability is crucial. Distributed and replicated storage provides the reliability guarantees that Google requires. Similarly, Amazon's front page user-customization makes, on average, 800 requests to a Berkeley DB database. An unavailable database loses customers.

Berkeley DB scales in terms of the amount of data it manages, the capabilities of the devices on which it runs, and the distance over which applications distribute data. Berkeley DB databases range from bytes to terabytes, with our largest installations reaching petabytes. Similarly, Berkeley DB runs on everything from small, handheld devices like Motorola cell phones to wireless routers [15] to large, multiprocessor servers. Berkeley DB's replication and high availability features are used both across backplanes and across the Internet.

The rest of this paper is organized as follows. In Section 2, we present the history of Berkeley DB. In section 3, we describe the Berkeley DB product family as it exists today. In section 4, we discuss how Berkeley DB addresses the widely varying needs of the environments in which it is embedded and the challenges this flexibility introduces. We conclude in Section 5.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 A Brief History of DB

Berkeley DB began life in 1991 as a dynamic linear hashing implementation [13] designed to replace the historic dbm [1], ndbm [5], and hsearch [2] APIs. Prior to its release as a library in the 4.4 BSD release, Keith Bostic of the UC Berkeley Computer Science Research Group (CSRG) designed an access-method independent API and Mike Olson, also at UC Berkeley, added a Btree implementation under that API. In 1992, UC Berkeley released 4.4BSD including the hash and Btree implementations backing the generic "db" interface that has become known as db-1.85.

In 1992, Olson and Seltzer published a paper describing a package called LIBTP [14]. LIBTP was a transactional implementation of db-1.85 that demonstrated impressive performance, relative to conventional client-server database systems. The implementation was a research prototype that was never released.

Db-1.85 enjoyed widespread adoption after its release with 4.4BSD. In particular, both the MIT Kerberos project and the University of Michigan LDAP project incorporated it. A group at Harvard released a minor patch version db-1.86 for the Kerberos team in 1996. In November of 1996, Seltzer and Bostic started Sleepycat Software to develop and market a transactional implementation of Berkeley DB. The transactional implementation was released in 1997 as Berkeley DB 2.0, the first commercial release of Berkeley DB.

The Sleepycat releases of Berkeley DB were made available under the Sleepycat Public License, which is now known as a dual license. The code is open source and may be downloaded and used freely. However, redistribution requires that either the package using Berkeley DB be released as open source or that the distributors obtain a commercial license from Sleepycat Software.

In 1999, Sleepycat Software released Berkeley DB 3.0, which transformed the original APIs into an object-oriented handle and method style interface. In 2001, Sleepycat Software released Berkeley DB 4.0, which included single-master, multiple-reader replication. This product, Berkeley DB/High Availability, offers outstanding availability, because replicas can take over for a failed master, and outstanding scalability, because read-only replicas can reduce master load. In 2006, Oracle acquired Sleepycat Software to round out its embedded database offerings. The Sleepycat products continue to be developed, maintained and supported by Oracle and are still available under a dual license.

3 Berkeley DB Today

The Berkeley DB product family today includes the original Berkeley DB library as well as an XML product (Berkeley DB XML) that layers an XML schema and Xquery and Xpath capabilities atop the original Berkeley DB library. Berkeley DB Java Edition (JE) provides functionality comparable to Berkeley DB in a 100% pure Java implementation that has been designed to work well in today's Java Virtual Machines. Figure 1 shows the Berkeley DB product family architecture.

The rest of this paper will focus on the Berkeley DB C-library. Other papers [6, 9] describe Berkeley DB XML and Berkeley DB Java Edition.

3.1 Overview

We begin this section with a brief overview of Berkeley DB, omitting many of the details that appear in other places [10]. We then discuss Berkeley DB's replication and high availability in greater detail. Section 4 then goes on to discuss the flexibility applications enjoy when they use Berkeley DB and the challenges introduced by this flexibility.

Berkeley DB provides high-performance persistent storage of key data pairs. Both keys and data are opaque byte-strings; Berkeley DB databases have no schema. The application that embeds Berkeley DB is responsible for imposing its own schema on the data. The advantage of this approach is that an application is free to store data in whatever form is most natural to it. If the application manipulates objects, the objects can be stored

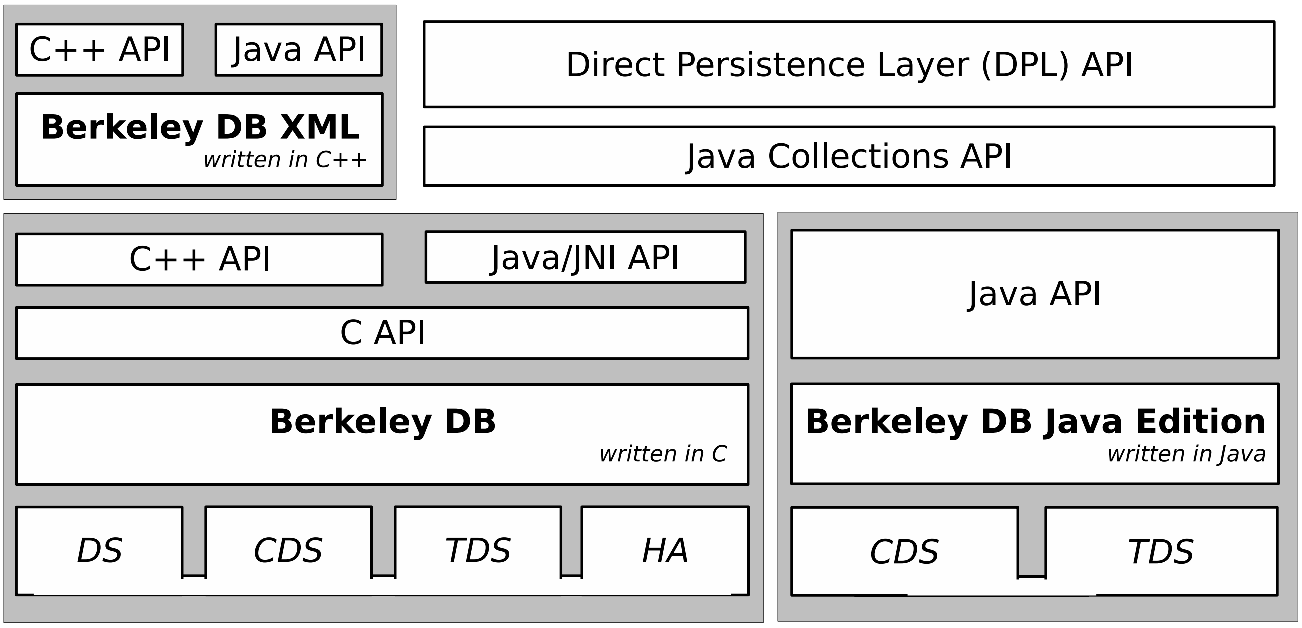


Figure 1: Berkeley DB Product Family: Berkeley DB consists of three products. The original transactional C-library key/value store is shown in the lower left corner, including APIs in Java and C++. Berkeley DB XML occupies the upper left corner, where it sits atop Berkeley DB. Berkeley DB Java Edition (JE) is in the lower right. Note that the Java Collections API and Direct Persistence Layer are APIs that sit atop either JE or the Java/JNI interface of the C-library.

directly in Berkeley DB; if the application manipulates relational rows, those too can be stored. In fact, different data formats can be stored in the same databases, as long as the application understands how to interpret the data items. This provides maximum application flexibility.

Berkeley DB provides several different data structures for indexing key values. The Btree index is by far the most widely used index type, but the library provides hash and queue indexes and a record-number-based index implemented on top of the Btree index. Applications *put* key/value pairs into databases, retrieve them via *get* calls, and remove them using *delete* methods. Indexes can require that all key values be unique or they can be allowed to accept duplicate values for the same key. All such policy decisions are left up to the application.

Applications access key/value pairs through handles on databases (somewhat akin to relational tables) or through *cursor* handles. Cursors are primarily used for iteration, representing a specific place within a database. Databases are implemented on top of the native file system; a file may contain one or more databases.

Transactional APIs provide the ability to wrap one or more operations inside a transaction, providing the ability to abort a sequence of operations and also providing recovery from application, operating system, or hardware failure. Unlike many other systems, database-level operations, such as database create, rename, and remove, can also be encapsulated in transactions.

3.2 Replication provides Scalability and High Availability

Following the basic design philosophy of Berkeley DB, the replication features provide mechanisms without specifying policies. The library provides the ability to create and manage any number of replicas, but lets the application control the degree of replication, how tightly synchronized the replicas are, how the replicas communicate, etc.

Berkeley DB replication is a log-shipping system. A replication group consists of a single *master* and one or more read-only *replicas*. All write operations, such as database create, key/value insert, update, or delete, and database destroy, must be processed transactionally by the master. The master sends log records to each of the replicas. The replicas apply log records only when they receive a transaction commit record. Therefore, aborted transactions introduce almost no load on the replicas; replicas will record log records from aborted transactions, but will not perform any work on their behalf.

If the master fails, one of the replicas can take over as master. In most cases, applications use the Berkeley DB paxos-compliant [8] election mechanism to select the new master, however, applications can choose to select a new master themselves. This latter approach is most frequently used when the application is running on top of a high availability substrate that includes master selection logic.

The Berkeley DB election algorithm is designed to prevent transaction loss. The replicas all send votes to each other indicating their application-specified priority, their maximum log record, and a unique tie-breaker. Each replica tallies the votes, selecting the winning site first based upon log records, then upon priority, and finally upon tie-breaker value. Thus, a low priority site with a complete set of log records will be elected before a higher priority site that is missing log records. After an election, each replica synchronizes with the master and then begins accepting and applying log records from the new master as it had done before the election.

Berkeley DB replication provides both scalability and high availability. It achieves scalability, because an application can deploy as many read-only replicas as necessary to scale with the read load. It delivers high availability, because sub-second elections with automatic failover and recovery provide the appearance of non-stop operation. Berkeley DB High Availability has been in widespread deployment for over five years in mission critical services, such as Google's Single Sign On service [12].

4 Flexibility Opportunities and Challenges

Configuration flexibility is critical for Berkeley DB, because it embeds in such a wide range of environments. The demands of a cell phone environment are dramatically different from those of a server farm running thousands of instances of a Berkeley DB application. Berkeley DB is successful in this huge range of environments by providing applications enormous flexibility in how they use the database. In this section, we discuss some of the most commonly used configuration settings; for more detailed discussion of configuration options, please see the Berkeley DB documentation [11].

There are three different ways to take advantage of Berkeley DB flexibility. First, Berkeley DB provides a range of build-time options that drive compilation of the library. Second, there are four different feature sets from which application developers can select; each feature set introduces trade-offs in functionality and performance. Third, there are run-time options that control the behavior of the library.

4.1 Compile Time Options

Compile-time configuration is provided by Berkeley DB's autoconfiguration [7] script. The two most common compile-time configuration options provide a small footprint build (*–enable-smallbuild*) and activate higher concurrency locking (*–enable-fine-grained-lock-manager*). These two options are designed for radically different environments and illustrate clearly the value of configurability.

The smallbuild configuration is designed for resource-constrained environments that need the power of the Berkeley DB library while consuming as few resources as possible. In fact, this configuration option arose from the constraints of a cell phone environment. When configured with smallbuild, the library contains only the Btree index (hash, queue, and record number are all omitted). It also omits replication, cryptography, statistics collection, and verification. The resulting library is approximately one-half megabyte and contains everything necessary to use transactional, recoverable btrees.

The fine-grained lock manager option is designed for large, highly concurrenct applications – think large servers running in a data center with hundreds or thousands of concurrent threads. Transactional applications must acquire locks to protect data while it is being read or written. These locks are maintained in shared memory data structures, whose integrity is maintained by acquiring and releasing mutexes while the data structures are being manipulated. The conventional Berkeley DB lock manager uses a single mutex to protect the entire lock shared memory region. Mutexes quickly become performance bottlenecks as they serialize access to memory. The advantage of this approach is that the application requires only a single mutex acquire/release pair for any lock operations. However, the disadvantage is that the mutex may be held for many thousands of instructions, limiting the degree of parallelism in the application, if it is lock intensive.

The fine-grained lock manager option trades off the number of mutex acquisition/release pairs against the advantage of having multiple threads or processes accessing shared data structures simultaneously. When this option is enabled, an application will acquire a larger number of mutexes, however, it will hold these mutexes for less time. Therefore, an application will observe reduced single-threaded performance, but better scalability as more threads of control are active simultaneously. This option should be considered carefully and used only when it provides significant benefit.

4.2 Feature Set Selection

The Berkeley DB library has four distinct feature sets as shown in Figure 1. The Data Store (DS) feature set is most similar to the original Berkeley DB 1.85 library. It provides high-performance storage for key/data pairs, but does not provide locking, transactions, or replication. The obvious benefit of this feature set is that it introduces the least overhead – no locking, no logging, no synchronous writes due to transaction commit. However, the disadvantages are also obvious – there can be no concurrent access by readers and writers and there is no recovery from any kind of failure. This approach is usually most suitable for temporary data storage.

The Concurrent Data Store (CDS) feature set addresses the lack of concurrency, but not recovery from failure. In CDS, the library locks at the API-layer, enforcing the constraint that there be either a single writer or multiple readers in the library, relieving the application of the burden of managing its own concurrency. CDS is most beneficial in the presence of concurrent workloads that are read-dominated with infrequent updates. It incurs slightly more overhead than DS, because it does acquire locks, but it incurs less overhead than the transactional feature set, because the library acquires only a single lock per API invocation, rather than a lock per page as is done with the transactional product.

Transactional Data Store (TDS) is currently the most widely used feature set. It provides both highly concurrent access and recovery from application, operating system, and hardware failure. The advantages to this configuration are numerous: multiple threads of control can access and modify the database simultaneously, data is never lost even after a failure, and the application can commit and abort transactions as necessary. However, this functionality comes at some performance cost. The library must log all updates, thereby increasing I/O. The library also obtains locks on database pages to prevent multiple threads from modifying the same page at the same time. Finally, the library will (normally) issue synchronous I/Os to commit transactions. Nonetheless, the benefits of the transactional feature set typically far outweigh these costs.

The High Availability (HA) feature set introduces the most complicated set of trade-offs. It provides the recoverability of TDS with the additional benefit that an application can continue running even after a site fails. It may seem that such functionality must incur a more significant performance penalty than TDS, but that is not always the case. In many cases, because many copies of the data exist, applications choose to commit transactions to remote replicas rather than require synchronous disk I/O. Commiting to the network is often faster than commiting to disk, so HA can offer *better* performance than TDS.

4.3 Runtime Configuration

Berkeley DB provides the greatest latitude at runtime where applications can make reliability/performance trade-offs and select settings most appropriate for a particular environment. There are far more runtime configuration options that can be discussed here, so we focus on those that applications most frequently use. First we talk about index structure selection and tuning, then discuss trade-offs between durability and performance, and conclude with a discussion of different concurrency management techniques.

As discussed above, Berkeley DB provides several different indexing data structures. In addition to selecting the appropriate index type, applications can select the underlying page size as well. Small pages improve concurrency, since Berkeley DB performs page-based locking; large pages typically improve I/O performance. The library will select a reasonable default page size that is well-matched to the underlying file system, but sometimes applications possess more information and can achieve better performance by explicitly setting a page size.

Another way that applications can tune indexes is by specifying application-specific key sorting and/or hash functions. By default, Berkeley DB uses lexicographic sorting on ordered keys and a simple XOR-based hash function. However, when applications are intimately familiar with the domain of their key values, an application can frequently select a sort or hash function more appropriate to the data.

The second area where applications frequently make critical configuration decisions is in trading off durability and performance. Berkeley DB must issue a synchronous disk I/O to achieve transactional durability on commit. The I/O is typically sequential, because it is a log write, but it still requires leaving high performance solid state memory and accessing a slow, mechanical device. Applications that can sacrifice durability guarantees can request that Berkeley DB commit asynchronously. In this case, the library will write a commit log record, but will not force it to disk. This means that upon failure, the library will still preserve transaction boundaries and leave the data in a transactionally consistent state, but some of the last transactions to commit may be lost. A slightly stronger form of this technique is also available; the library will write the log records to the operating system's buffer cache, but will not force the disk write. This avoids nearly all of the cost of the log write and protects against transaction loss when the application crashes, but the system continues to run. As discussed above, avoiding the disk write at commit is frequently used when an application is already replicating its data to other sites.

Berkeley DB allows applications to take disk-write avoidance to the extreme: applications can run completely in memory. In a pure in-memory environment, all databases, log files, etc are stored in memory. Obviously, if the application crashes, all data is lost, however, in some environments sacrificing durability for higher performance is attractive. For example, consider a router. While the router is running, performance is critical, and even if some components in the router fail, as long as some components are running, availability is crucial (i.e., you do not want to drop connections). However, if the router crashes, restoring connection state information after a restart provides no value. In such a configuration, Berkeley DB's pure in-memory configuration is attractive. Data and log records are never written to disk, but both are replicated across multiple hardware instances. In this manner, the system provides both outstanding availability and outstanding performance.

Concurrency control techniques are another mechanism that can dramatically affect performance and scalability. By default, Berkeley DB uses conventional two-phase locking (2PL) [4]. In 2PL, the library acquires a lock of the appropriate mode (read or write) before accessing or modifying data, holding the locks until commit. The two phases of 2PL refer to a first phase during which the library acquires locks and a second phase during which it releases locks. In Berkeley DB, as in most systems, the first phase lasts during the entire duration of the transaction and the second phase is fully encapsulated at commit or abort time.

Two-phase locking usually works quite well, but can limit concurrency during database traversal, because a database traversal will ultimately lock the entire database. When such traversals are common, multiversion concurrency control (MVCC) [3] provides significant benefits.

In MVCC, writers make copies of database pages rather than acquiring locks on them. While such writing transactions are active, readers can continue to access the original copies of the pages. Thus, database traversal can proceed even in the presence of update transactions. MVCC provides increased concurrency for some workloads at the expense of increased memory consumption and data copies. Berkeley DB provides both two-phase locking and multi-version concurrency control.

4.4 Challenges

Berkelely DB's tremendous flexibility introduces a set of unique challenges. Features that are obviously desirable for one environment may be completely unacceptable for another. For example, some customers wonder why Berkeley DB does not assume TCP/IP as its replication communication protocol. While TCP/IP makes sense in some environments, it is not acceptable in an environment where Berkeley DB is replicated across a backplane. Historically, Berkeley DB has left any and all such policy decisions to the application. Unfortunately, that placed a heavy burden on the application developer. An alternative approach is to offer such solutions, allowing developers to override them when necessary. This philosophy is embodied in the recently-introduced *Replication Framework*.

The Replication Framework makes several policy decisions regarding how replication is managed. For example, TCP/IP is the communication infrastructure and threading is provided via pthreads. The advantage is that application developers write significantly less code. The disadvantage is that these decisions limit some of the library's flexibility. This is a tension that will always exist in Berkeley DB as long as it continues to be an appropriate solution from cell phones to servers.

The fact that Berkeley DB is a library that links into an application introduces a unique set of problems. The library design must be correct regardless of the application developer's choices for surrounding components. For example, the library must work correctly regardless of the choice of thread package, file system, mutex implementation, etc.

Adding open-source to the library equation introduces additional issues. Because Berkeley DB is available in source-code form, it becomes an obvious choice for developers working on new or unusual hardware platforms. Thus, portability is a requirement. When customers are compiling your software, they will use any compiler available, so the software must be portable across all compilers. Although languages have been "standardized" for years, each compiler introduces its own idiosyncracies and constraints. For example, a recent release of glibc made Berkeley DB fail to compile, because the system includes contain a macro definition for open, and multiple Berkeley DB data structures contain an open method.

Code quality also takes on a significant role when customers can (and do) read your source code. A poorly expressed comment can jeopardize sales when potential customers read the code and question the comments. Even comments that allude to not-yet-implemented features can be problematic. Nothing beats open source for encouraging rigorous adherence to coding standards and clear, correct commenting.

Another constant tension concerns what features are best provided by the library and what features are better provided by the embedding application. As discussed by Perl and Seltzer [12], *master leases* provide a guaranteed ability to perform reads that never return stale data. They conclude that while Google was able to implement master leases in the application, it would have been better if such functionality were available in the library. In a similar vein, the Berkeley DB replication API does not provide a mechanism to manage replication group membership; this capability resides in the replication framework. Nonetheless, it would be useful functionality to applications that cannot be constrained by other choices in the replication framework.

In general, Berkeley DB incorporates new features when there is clear demonstrated customer demand from multiple market segments. Such decision making requires flexibility in both the library, the Berkeley DB designers, and those application developers using Berkeley DB.

5 Conclusions

Berkeley DB is a dramatically different library than its db-1.85 ancestor. Its roots are clearly visible, but it has matured gracefully over time. What began life as a simple key/value storage engine now provides mission critical data storage for many key Internet infrastructure services.

6 Bibliography

References

[1] AT&T, DBM(3X), *UNIX Programmer's Manual, Seventh Edition*, Volume 1, January", 1979.

[2] AT&T, HSEARCH(BA-LIB), *UNIX System User's 's Manual, System V*, Volume 3:506–508, 1985.

[3] P. Bernstein and N. Goodman, "Concurrency control algorithms for multiversion database systems", *Proceedings of the first ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 209–215, ACM Press, Ottawa, Canada, 1982.

[4] P. Bernstein and V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley Publishing Company, 1987.

[5] Computer Systems Research Group, NDBM(3), *4.3 BSD UNIX Programmer's Manual Reference Guide*, University of California, Berkeley", 1986.

[6] P. Ford "Berkeley DB XML: An Embedded XML Database", *XML.com*, O'Reilly and Associates, May, 2003.

[7] GNU. autoconf. http://www.gnu.org/software/autoconf.

[8] L. Lamport, "The Part-Time Parliament", *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[9] J. Menard, "Building Applications with Berkeley DB Java Edition", *Java Developer's Journal*, Sys-Con Media, September 2004.

[10] M. Olson and K. Bostic and M. Seltzer, "Berkeley DB", *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, Monterey, CA, June 1999.

[11] Oracle Corporation. Berkeley DB Documentation. http://www.oracle.com/technology/documentation/berkeley-db/db/index.html.

[12] S. Perl and M. Seltzer, "Data Management for Internet-Scale Single-Sign-On", *Proceedings of the 3rd Workshop on Real, Large Distributed Systems (WORLDS '06)*, Seattle, WA, November 2006.

[13] M. Seltzer and O. Yigit, "A New Hashing Package for UNIX", *Proceedings of the 1991 Winter USENIX Conference*, 173–184, USENIX Association, Dallas, TX, 1991.

[14] M. Seltzer and M. Olson "LIBTP: Portable, Modular Transactions for UNIX", *Proceedings of the 1992 Winter USENIX Conference*, 9–26, "USENIX Association, San Francisco, CA, 1992.

[15] R. Van Tassle and M. Richardson. Wireless networking & Berkeley DB. *Dr. Dobbs Journal*, 27(1), CMP Media, January 2002.

SQL Anywhere: An Embeddable DBMS

Ivan T. Bowman Peter Bumbulis Dan Farrar Anil K. Goel Brendan Lucier
Anisoara Nica G. N. Paulley John Smirnios Matthew Young-Lai
Sybase iAnywhere

Abstract

We present an overview of the embeddability features of SQL *Anywhere, a full-function relational database system designed for frontline business environments with minimal administration.* SQL *Anywhere supports features common to enterprise-class database management systems, such as intra-query parallelism, materialized views,* OLAP *functionality, stored procedures, triggers, and hot failover.* SQL *Anywhere can serve as a high-performance workgroup server, an embedded database that is installed along with an application, or as a mobile database installed on a handheld device that provides full database services, including two-way synchronization, to applications when the device is disconnected from the corporate intranet. We illustrate how* SQL *Anywhere's embeddability features work in concert to provide a robust data management solution in zero-administration environments.*

1 Introduction

Database systems have become ubiquitous across the computing landscape. This is partly because of the basic facilities offered by database management systems: physical data independence, ACID transaction properties, a high-level query language, stored procedures, and triggers. These facilities permits sophisticated applications to 'push' much of their complexity into the database itself. The proliferation of database systems in the mobile and embedded market segments is due, in addition to the features above, to the support for two-way database replication and synchronization offered by most commercial database management systems. Data synchronization technology makes it possible for remote users to both access *and update* corporate data at a remote, off-site location. With local (database) storage, this can be accomplished even when disconnected from the corporate network.

In this paper, we describe a wide range of technologies that permit SQL Anywhere [2] to be used in embedded and/or zero-administration environments. In Section 2, we outline architectural features of SQL Anywhere that are helpful, if not essential, in embedding the database with a specific application to provide data managment services. In Section 3, we briefly describe self-management technologies within the SQL Anywhere server to enable its operation in zero-administration environments. These self-managing technologies are fundamental components of the server, not merely administrative add-ons that assist a database administrator in configuring the server's operation—since in embedded environments there is often no trained administrator to do so. It is important to note that these technologies work in concert to offer the level of self-management and adaptiveness that embedded application software requires. It is, in our view, impossible to achieve effective self-management by considering these technologies in isolation.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Embeddability-enabling technologies

SQL Anywhere was designed from the outset to offer self-management features permitting its deployment as an embedded database system. The deployment of a SQL Anywhere server and database instance can be integrated with the application's installation procedure; both database files and executable program files can be renamed to fit the application's requirements. Moreover, the server can be configured so that its operation is transparent. Below, we highlight some of the embeddability features that enable SQL Anywhere to be used as an embedded database.

Autostart and autostop. A SQL Anywhere server instance can be started by a simple client API call from the application over a shared-memory connection on the same physical hardware. Once the server is started, additional databases can be started or stopped via SQL statements sent from any application. In turn, a server or database can be configured to shut down automatically when the last connection disconnects.

Database storage utilizes the native file system. SQL Anywhere databases are stored as ordinary OS files and can be managed with the file utilities provided by the operating system. Each database consists of a main database file, a separate transaction log file, and up to 12 additional files that can be placed on the same filesystem or spread across several. Raw partitions are not supported. The benefit of this simplicity is *deployment flexibility*. To image copy a database, one simply copies all of its associated files; execution of a copy utility program is not required. Database files are portable amongst all supported platforms, including Windows CE devices, and even if the machines are of different CPU architectures. This flexibility makes database deployment, application development, and problem determination, particularly to handheld devices and/or remote locations, significantly easier.

The transaction log contains logical operations. Unlike write-ahead logging approaches such as Aries [6] that log physical operations, SQL Anywhere's transaction log contains logical operations. In addition to providing the basis for database synchronization operations, the forward transaction log can be translated into equivalent SQL statements, which can then be applied to any other database instance with the same schema. This architecture offers application vendors considerable flexibility when performing problem determination and, if necessary, database reconstruction at remote locations.

Integrated security. SQL Anywhere provides full end-to-end security with 128-bit strong encryption of database tables, files, and communications streams between the application and the database, as well as the MobiLink synchronization stream. SQL Anywhere offers built-in user authentication (including support for Kerberos), and can integrate with third-party authentication systems. Client applications can verify the identity of a server using digital signatures or signed certificates. In addition, application vendors can *authenticate* a server, which can prevent usage of the server by other (unauthorized) applications entirely, or restrict their capabilities. SQL Anywhere also offers FIPS-certified encryption via a separately licensed security option. Further, SQL Anywhere supports encryption of business logic stored in the database to prevent reverse engineering of the OEM application logic (stored procedures, views, triggers and so on).

Application profiling. To support problem determination and analysis of application performance problems in the field, SQL Anywhere contains built-in application profiling that can obtain a detailed trace of all server activity, including SQL statements processed, performance counters, and contention for rows in the database. This trace information is captured as an application runs, and is transferred via a TCP/IP link into any SQL Anywhere database, where it can be analyzed. This flexible architecture allows for the trace to be captured with a focus on convenience (by storing the trace in the same database that generated it) or on performance (by storing the trace data on a database on a separate physical machine). The architecture also permits the Application Profiling tool to analyze and make recommendations, including index recommendations, for databases on mobile devices running Windows CE.

Quiet operation. Both the personal and the network SQL Anywhere server can be configured to operate in 'quiet' mode. During quiet operation, no console log is displayed, and the server does not issue startup messages or issue prompts for service. This permits the OEM application to be in complete control of the user interface for the application.

Active database support. SQL Anywhere supports events, using either timer-based or event-based scheduling of event handlers (stored procedures), which can permit automatic handling of a variety of conditions, from 'disk full' errors to application disconnection. The server's built-in SMTP support permits event handlers to issue email notification of events where desirable.

Support for web services. SQL Anywhere contains a built-in HTTP server that allows the server to function as a web server, as well as to permit access to web services in other SQL Anywhere databases and standard web services available over the Internet. SOAP is the standard used for this purpose, but the built-in HTTP server in SQL Anywhere also supports standard HTTP and HTTPS requests from client applications. As a consequence, it is possible to embed an entire application within a database instance, using the active database components supported by the server, along with the server's built-in XML capabilities. If packaged in this way, deploying the 'application', with its database, is as simple as deploying the database instance itself; all that is required for a user interface is a web browser.

3 Self-management technologies

3.1 Dynamic buffer pool management

When a database system is embedded in an application as part of an installable package, it cannot normally use all the machine's resources. Rather, it must co-exist with other software and system tools whose configuration and memory usage vary from installation to installation, and from moment to moment. In this environment, it is difficult to predict the system load or the amount of memory that will be available at any point in time.

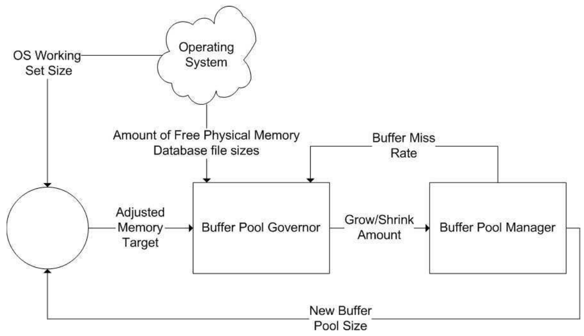


Figure 1: Cache sizing feedback control

Consequently, SQL Anywhere uses the following approach to buffer pool management: rather than attempting to 'tune' buffer pool memory in isolation, the server tunes buffer pool allocation to fit the overall system requirements. It does this by using a feedback control mechanism, using the OS working set size, the amount of free physical memory, and the buffer miss rate as the inputs (see Figure 1). The OS working set size, which is polled every minute, is the operating system's amount of real memory in use by the process. Using this feedback control loop, the server's buffer pool grows or shrinks on demand, depending on system-wide resource usage and the memory management policy of the operating system. This adjustment can be done very efficiently on some operating systems that permit address space to be allocated to a process independent of backing physical memory. The variability of the buffer pool size has implications for query processing. Queries must adapt to execution-time changes in the amount of available physical memory (see Section 3.3).

Heaps. A novel feature of SQL Anywhere is that the buffer pool is a single heterogeneous pool of all types of pages: table pages, index pages, undo and redo log pages, bitmaps, free pages, and heap pages. To support efficient buffer pool management, all page frames are the same size.

Data structures created and utilized for query processing, including hash tables, prepared statements, cursors, and similar structures are allocated inside pages in a heap. When a heap is not in use—for example, when the server is awaiting the next FETCH request from the application—the heap is 'unlocked'. Pages in unlocked heaps can be stolen and used by the buffer pool manager for other purposes, such as caching table or index pages, as required. When this happens, the stolen pages are swapped out to the temporary file. To resume the processing of the next request when it arrives, the heap is re-locked, pinning its pages in physical memory. A pointer swizzling technique (cf. reference [4]) is used to reset pointers in pages forced to be relocated during re-locking.

Page replacement strategy. As described above, all page frames in the buffer pool are the same size, and can be used for various purposes. Each type of page has different usage: heap pages, for example, are local to a connection, while table pages are shared. The buffer pool's page replacement algorithm must be aware of the differences in usage as well as those in reference locality in page access patterns. For example, it must recognize that adjacent references to a single page during a table scan are different from other reference patterns. SQL Anywhere uses a modified generalized 'clock' algorithm [9] for page replacement. In addition, we have implemented techniques to reduce the overhead of the clock algorithm, yet quickly recognize page frames that can be reused.

3.2 Self-managing statistics

SQL Anywhere has used query feedback techniques since 1992 to automatically gather statistics during query processing. Rather than require explicit generation of statistics, by scanning or sampling persistent data, the server automatically collects statistics as part of query execution. Subsequent work by other researchers [1, 12] has also exploited this notion of collecting statistics as a by-product of query execution.

In its early releases, SQL Anywhere computed frequent-value statistics from the evaluation of equality and Is Null predicates, and stored these statistics persistently in the database for use in the optimization of subsequent queries. Later, support was added for inequality and LIKE conditions. An underlying assumption of this model is that the data distribution is skewed; values that do not appear as a frequent-value statistic are assumed to be in the 'tail' of the distribution.

Today, SQL Anywhere uses a variety of techniques to gather statistics and maintain them automatically. These include index statistics, index probing, analysis of referential integrity constraints, three types of single-column self-managing histograms, and join histograms.

3.2.1 Histogram implementation

SQL Anywhere uses self-tuning histograms whose bucket pools expand and contract dynamically as column value distribution changes are detected. The structure of bucket boundaries shifts over time to minimize error and allocate more resolution where it is needed. As is typical, we use the *uniform distribution assumption* when interpolating within a bucket. SQL Anywhere histograms combine traditional buckets with frequent-value statistics, known as 'singleton buckets'. Singletons are implemented as buckets ranging over single values, and are interleaved with traditional buckets in the histogram structure. A value that represents at least 1% of the column data or appears in the 'top N' most frequent values is saved as a singleton bucket. The number of singletons retained in any histogram depends on the size of the table and the column's distribution, but lies in the range [0,100].

For efficiency and simplicity, the same infrastructure is used for almost all data types. An order-preserving hash function, whose range is a double-precision floating point value, is used to derive the bucket boundaries on these data types. The exceptions are longer string and binary data types, for which SQL Anywhere uses a different infrastructure that dynamically maintains a list of observed predicates and their selectivities. In addition, not only are buckets created for entire string values, but also for LIKE patterns intended to match a 'word' somewhere in the string.

Statistics collection during query processing. Histograms are created automatically when data is loaded into the database using a LOAD TABLE statement, when data is updated using INSERT, UPDATE, and DELETE statements, when an index is created, or when an explicit CREATE STATISTICS statement is executed. A modified version of Greenwald's algorithm [5] is used to create the cumulative distribution function for each column. Our modifications significantly reduce the overhead of statistics collection with a marginal reduction in quality. In addition to data change operations, histograms are automatically updated during query processing. During normal operation, the evaluation of (almost) any predicate over a base column can lead to an update of the histogram for this column.

Join histograms are computed on-the-fly during query optimization to determine the cardinality and data distribution of intermediate results. Join histograms are created over single attributes. In cases where the join condition is over multiple columns, a combination of existing referential integrity constraints, index statistics, and density values are used to compute and/or constrain join selectivity estimates.

A variety of additional statistics are automatically maintained for other database objects. For stored procedures used in table functions, the server maintains a summary of statistics for previous invocations, including total CPU time and result cardinality. A moving average of these statistics is saved persistently in the database for use by the optimizer for subsequent queries. In addition, statistics specific to certain values of the procedure's input parameters are saved and managed separately if they differ sufficiently from the moving average. Index statistics, such as the number of distinct values, number of leaf pages, and clustering statistics, are maintained in real time during server operation. Table statistics, in particular the percentage of a table resident in the buffer pool, are also maintained in real time and used by the cost model when computing a table's access cost.

3.3 Query processing

In our experience, there is little correlation between application or schema complexity, and the database size or deployment platform. Developers tend to complicate, rather than simplify, application design when they migrate applications to business front-lines, even when targeting platforms like hand-held devices with few computing resources. It is usually only the user-interface that is re-architected because of the input mode differences on such devices.

This complexity makes sophisticated query processing an essential component of SQL Anywhere. As described in Section 3.1, the operating characteristics of the server can change from moment to moment. In mixed-workload systems with complex queries, this flexibility demands that query processing algorithms adapt to changes in the amount of memory they can use. It also means that the query optimizer must take the server state into account when choosing access plans.

3.3.1 Query optimization

SQL Anywhere (re)optimizes a query[1] at each invocation. There are two broad exceptions to this. The first class of exceptions is simple DML statements, restricted to a single table, where the cost of optimization approaches the cost of statement execution. In such cases, these statements bypass the cost-based optimizer, and are optimized heuristically. The second class is statements within stored procedures and triggers. For these statements, access plans are cached on an LRU basis for each connection. A statement's plan is only cached, however, if the access plans obtained by successive optimizations of that statement during a 'training period' are identical. After the training period is over, the cached plan is used for subsequent invocations. However, to ensure the plan remains 'fresh', the statement is periodically verified at intervals taken from a decaying logarithmic scale.

Re-optimization of each query means that optimization cost cannot be amortized over many executions. Optimization must therefore be cheap. One of several techniques used to reduce optimization cost is to limit the size of the optimizer's search space. The SQL Anywhere optimizer uses a proprietary branch-and-bound, depth-first search enumeration algorithm [8] over left-deep processing trees[2]. Depth-first search has the significant advantage of using very little memory; in fact, much of the state information required by the algorithm can be kept on the processor stack. In this approach, enumeration and cost estimation are interleaved.

[1]In this context we use the term 'query' to refer to not only queries but also to INSERT, UPDATE, and DELETE statements.

[2]Left-deep trees are used except for cases involving complex derived tables or table expressions containing outer joins.

The enumeration algorithm first determines a heuristic ranking of the tables involved in the join strategy. In addition to enumerating tables or table functions, the algorithm also enumerates materialized views matching parts of the query, and complex subqueries by converting them into joins on a cost basis [7]. Join and index physical operators are also enumerated during join strategy generation which makes possible costing plans with intra-query parallelism. By considering tables in rank order, the enumeration algorithm initially (and automatically) defers Cartesian products to as late in the strategy as possible. Hence it is likely that the first join strategy generated, though not necessarily optimal, will be one with a 'reasonable' overall cost, relative to the entire search space. The algorithm is branch-and-bound in the sense that a partial join strategy is retained only if its cost is provably less than the cost of the best complete join strategy discovered thus far.

An interesting characteristic of the branch-and-bound enumeration algorithm is the method by which the search space is pruned during join strategy generation. The algorithm incrementally costs the prefix of a join strategy and backtracks as soon as the cost of an intermediate result exceeds that of the cheapest complete plan discovered thus far. Since any additional quantifiers can only add to the plan's cost, no join strategy with this prefix of quantifiers can possibly be cheaper and the entire set of such strategies can be pruned outright. This pruning is the essence of the algorithm's branch-and-bound paradigm. It is described in detail in reference [3].

Additional care must be taken when analyzing cost measures for an intermediate result. For example, a significant component of any plan's cost concerns its buffer pool utilization. However, measures such as buffer hit ratios can be accurately estimated only with regard to a *complete* execution strategy, since in a fully-pipelined plan the most-recently used pages will be from those tables at the root of the processing tree. Nonetheless, it is possible to estimate the cost of computing an intermediate result based on a *very* optimistic metric: assume that half the buffer pool is available for each quantifier in the plan. Clearly this is nonsense with any join degree greater than 1. However, the point is not to accurately cost intermediate results, but to quickly prune grossly inefficient strategies from the search space.

Another notable characteristic of the join enumeration algorithm involves the control strategy used for the search [8]. A problem with join enumeration using a branch-and-bound approach with early halting is that the search effort is not well-distributed over the entire search space. If a small portion of the entire space is visited, most of the enumerated plans will be very similar. SQL Anywhere addresses this problem by employing an optimizer governor [10] to manage the search. The governor dynamically allocates a quota of search effort across sections of the search space to increase the likelihood that an efficient plan is found. This quota is unevenly distributed across similar join strategies so that more effort is spent on heuristically higher-ranked strategies. Initial quota can be specified within the application, if desired, allowing fine-grained tuning of the optimization effort spent on each statement.

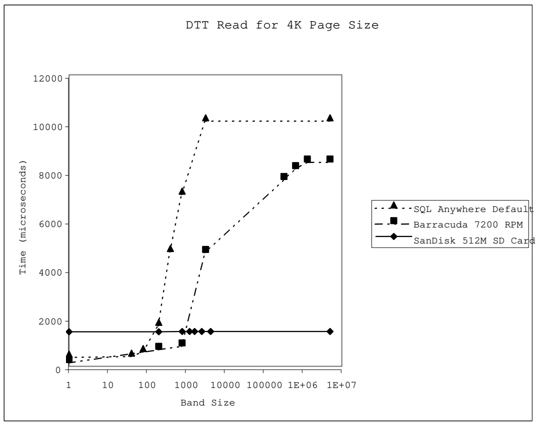


Figure 2: DTT models

Disk transfer time model. SQL Anywhere uses a Disk Transfer Time (DTT) model [4] to estimate a query's expected I/O cost. A 'generic' model is used by default. It has been validated using a systematic testing framework over a variety of machine architectures and disk subsystems. While the default DTT model performs well for a range of hardware devices, SQL Anywhere also provides a mechanism to calibrate using a specific hardware device. Further, the calibrated model can be saved in a file and loaded into multiple other databases. This approach is very useful in cross-platform development where the development platform is significantly different from the deployment platform.

The DTT function summarizes disk subsystem behaviour with respect to an application (in this case, the SQL Anywhere server). The DTT function models the amortized cost of reading one page randomly over a *band size* area of the disk. If the band size is 1, the I/O is sequential; otherwise, it is random. Significantly larger band sizes increase the average cost of each retrieval because of a higher probability of each retrieval requiring a seek, and the increase in the travel time of the disk arm to reach the correct cylinder. Figure 2 illustrates the default and calibrated DTT taken from two different hardware configurations.

3.3.2 Adaptive query execution

SQL Anywhere's query optimizer can automatically annotate a chosen query execution strategy with alternative plan operators that offer a cheaper execution technique if either the optimizer's choices are found to be suboptimal at run-time, or the operator requires a low-memory strategy at the behest of the memory governor (see below). For example, the optimizer may construct an alternative index-nested loops strategy for a hash join, in case the size of the build input is considerably smaller than that expected at optimization time. Hash join, hash-based duplicate elimination, and sorting are examples of operators that have low-memory execution alternatives. A special operator for execution of RECURSIVE UNION is able to switch between several alternative strategies, possibly using a different one for each recursive iteration.

To ensure that SQL Anywhere maintains a consistent memory footprint, in-memory data structures used by query processing operators are allocated within heaps that are subject to page replacement within the buffer pool. Moreover, as the buffer pool can shrink during query execution, memory-intensive query execution operators must be able to adapt to changing memory conditions.

Each task, or unit-of-work, within the SQL Anywhere server is given a quota of available memory by the server's memory governor. There are two quotas computed for each task:

- a *hard* memory limit: if exceeded, the statement is terminated with an error.
- a *soft* memory limit, which the query processing algorithms used for the statement should not exceed, if at all possible. Approaching this limit may result in the memory governor requesting query execution operators to free memory if possible.

The memory governor controls query execution by limiting memory consumption for a statement to not exceed the soft limit. For example, hash-based operations in SQL Anywhere choose a number of buckets based on the expected number of distinct hash keys; the goal is to have a small number of distinct key values per bucket. In turn, buckets are divided uniformly into a small, fixed, number of partitions. The number of partitions is selected to provide a balance between I/O behaviour and fanout. During the processing of the hash operation, the governor detects when the query plan needs to stop allocating more memory—that is, it has exceeded the soft limit. When this happens, the partition with the largest number of rows is evicted from memory. The in-memory rows are written out to a temporary table, and incoming rows that hash to the partition are also written to disk.

By selecting the partition with the most rows, the governor frees up the most memory for future processing, in the spirit of other documented approaches in the literature [11]. By paying attention to the soft limit while building the hash table on the smaller input, the server can exploit as much memory as possible, while degrading adaptively if the input does not fit completely in memory. In addition, the server also pays attention to the soft limit while processing the *probe* input. If the probe input uses memory-intensive operators, their execution may compete with the hash join for memory. If an input operator needs more memory to execute, the memory governor evicts a partition from the consuming operator's hash table. This approach prevents an input operator from being starved for memory by a consumer operator.

Adaptive intra-query parallelism. SQL Anywhere can assign multiple threads to a single request, thus achieving intra-query parallelism. There is no static partitioning of work amongst the threads assigned to a query. Rather, a parallel query plan is organized so that any active thread can grab and process a small unit of work from anywhere in the data flow tree. The unit of work is usually a page's worth of rows. It is not important which thread executes a given unit of work; a single thread can execute the entire query if additional threads are not available. This means that parallel queries adapt gracefully to fluctuations in server load.

4 Conclusions

Virtually every new feature implemented in SQL Anywhere is designed to be adaptive or self-managing. In addition to the embedded and self-management technologies described above, SQL Anywhere also includes a variety of tools and technologies that are useful during the development of a database application, including an index selection utility called the *Index Consultant*, graphical administration and modeling tools, and a full-function stored procedure debugger. Going forward, we will continue to develop autonomic features that provide greater degrees of self-tuning and self-management.

References

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In ACM SIGMOD *International Conference on Management of Data*, pages 181–192, Philadelphia, Pennsylvania, May 1999.

[2] I. T. Bowman, P. Bumbulis, D. Farrar, A. K. Goel, B. Lucier, A. Nica, G. N. Paulley, J. Smirnios, and M. Young-Lai. SQL Anywhere: A holistic approach to database self-management. In *Proceedings, ICDE Workshops (Self-Managing Database Systems)*, pages 414–423, Istanbul, Turkey, Apr. 2007. IEEE Computer Society Press.

[3] I. T. Bowman and G. N. Paulley. Join enumeration in a memory-constrained environment. In *Proceedings, Sixteenth* IEEE *International Conference on Data Engineering*, pages 645–654, San Diego, California, Mar. 2000.

[4] A. K. Goel. *Exact Positioning of Data Approach to Memory Mapped Persistent Stores: Design, Analysis and Modelling*. PhD thesis, University of Waterloo, Waterloo, Ontario, 1996.

[5] M. Greenwald. Practical algorithms for self-scaling histograms or better than average data collection. *Performance Evaluation*, 20(2):19–40, June 1996.

[6] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM *Transactions on Database Systems*, 17(1):94–162, Mar. 1992.

[7] A. Nica. System and methodology for cost-based subquery optimization using a left-deep tree join enumeration algorithm. US Patent 2004/0220923, Nov. 2004.

[8] A. Nica. System and methodology for generating bushy trees using a left-deep tree join enumeration algorithm. US Patent 7,184,998, Feb. 2007.

[9] A. J. Smith. Sequentiality and prefetching in database systems. ACM *Transactions on Database Systems*, 3(3):223–247, Sept. 1978.

[10] M. Young-Lai. Database system with methodology for distributing query optimization effort over large search spaces. US Patent 6,807,546, Oct. 2004.

[11] H.-J. Zeller and J. Gray. An adaptive hash join algorithm for multiuser environments. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 186–197, Brisbane, Australia, Aug. 1990.

[12] Q. Zhu, B. Dunkel, W. Lau, S. Chen, and B. Schiefer. Piggyback statistics collection for query optimization: Towards a self-maintaining database management system. *Computer Journal*, 47(2):221–244, 2004.

Thinking Big About Tiny Databases

Michael J. Franklin
UC Berkeley
franklin@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

Abstract

Work on early tiny database systems, like TinyDB [17] and Cougar [23] has shown that a declarative approach can provide a powerful and easy to use interface for collecting data from static sensor networks. These early systems, however, have significant limitations; in particular, they are useful only for low-rate data collection applications on static sensor networks and they don't integrate well with existing database and IT tools. In this paper, we discuss recent research that has addressed these limitations, showing that similar "tiny database thinking" can provide solutions to much bigger problems, including network protocol specification and mobile and high data rate signal-oriented data processing.

1 Introduction

Sensornets – collections of inexpensive, battery powered and wirelessly networked computers with sensing hardware for measuring light, temperature, vibration, sound, and a variety of other parameters – promise to measure the world at remarkably fine granularity. Touted applications include precision agriculture (monitoring the growth of individual plants), preventative maintenance of individual pumps or pipes in industrial settings, ecological monitoring of habitats, soil and water quality, and so on.

As with traditional enterprise database management systems, sensor network databases, like TinyDB [17] and Cougar [23] have proven to be a powerful tool for deploying such applications. In particular, these systems provide high level languages that allow users to specify what data they would like to receive without worrying about low-level details regarding how that data is collected or processed. In particular, details such as network and power management, time synchronization, and where data processing should be performed is completely hidden from the end user, greatly simplifying deployment.

Early "tiny databases" were developed several years ago. Since then, the idea of high level, declarative languages for sensor network programming has been refined in several ways. In this paper, we review a number of such refinements we have been working on in our research groups, including:

Declarative network programming. Early sensornet database systems provided declarative query languages for traditional database-like tasks of specifying data of interest to the user. We have since shown that recursive query languages can be used much more broadly throughout sensor network software infrastructure, resulting in code that is radically simpler to write and maintain than equivalent code in a traditional programming language [5]. We have demonstrated that a wide variety of ad-hoc sensor network protocols can be specified declaratively in

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

a very compact way, compiling down to efficient code that runs in a small footprint. The declarative approach relieves the programmer from many complexities that arise at both the networking and data processing layers, while raising a number of opportunities for innovation in both areas.

Sophisticated signal-oriented programming languages. The programming interface provided by TinyDB and Cougar was very simple, and allowed only basic relational operations of the sort that are common in business processing applications. This is insufficient for many sensor network applications which require more sophisticated signal-oriented processing. For example, in many industrial monitoring applications where vibrational analysis is used to detect malfunctions or failures, frequency transforms are used to measure the energy in certain frequency bands that characteristically show increased energy during failures. A major thrust of our recent research involves extending the declarative programming model to allow such sophisticated processing to be expressed in the same language as basic data processing operations.

Support for large-scale sensing applications. Early systems ignored issues related to how sensor networks integrate into a larger information processing ecosystem. One focus of our recent work has been on the integration of sensor data with other data sources and the processing of "roll up"-style aggregation queries over data from diverse collections of sensors.

Support for mobility and intermittent connectivity. Mobile sensor networks offer the potential to monitor much larger geographic areas than fixed networks. For example, by mounting accelerometers that can detect potholes and other road surface anomalies on a small number of cars, it is possible to monitor hundreds or thousands of miles of roads. Such mobile networks require a different software infrastructure from that provided by systems like TinyDB and Cougar, as mobile nodes often lack always-on network connections, requiring special support for intermittent connectivity.

Before describing our research in these areas in detail, we first summarize the basic declarative programming model developed in TinyDB and other tiny database systems.

2 Early Tiny Databases

The typical usage model for early sensor network databases is as follows: a collection of static sensor nodes is placed in some remote location, each of which is pre-programmed with the database software. These nodes report data wirelessly (often over multiple radio hops) to a nearby "basestation" – typically a laptop-class device with an Internet connection, which then relays data to a server where data is stored, visualized, and browsed.

Users interact with the system by issuing queries at the basestation, which in turn broadcasts queries out into the network. Queries are typically disseminated via flooding, or perhaps using some more clever gossip based dissemination scheme (e.g., Trickle [14]). As nodes receive the query, they begin processing it. The basic programming model is data-parallel: each node runs the same query over data that it locally produces or receives from its neighbors For example, a simple query that asks nodes to report when their temperature readings go above some threshold looks as follows:

```
SELECT nodeid, temp
 FROM sensors
 WHERE temp > thresh
 SAMPLE PERIOD 1s
```

Here, `sensors` is a "virtual table" containing one field per type of sensor available. By "virtual", we mean that it is never actually materialized in the memory of the sensor – instead, new rows are created in by sampling the sensors at the rate specified in the `SAMPLE PERIOD` clause, and those rows are transmitted off of the device before the next row is produced. Each node produces rows corresponding to its own local `nodeid`. Note that the evaluation of the `temp` $> thresh$ predicate can clearly be done on each individual node in this case, but that for more complicated predicates – e.g., that depend on the values of other nodes – in-network computation may not be as trivial. Models for such in-network processing are the focus of much work in the sensor network data processing community – see, for example, work on support for in-network aggregation of data from multiple nodes [16] and in-network join processing [2, 1].

When a node has some data to transmit, it relays it to the basestation using a so-called *tree-based routing protocol*. The idea with these protocols is that nodes arrange themselves into a tree rooted at the basestation. This tree is formed by having the basestation periodically broadcast a beacon message. Nodes that hear this beacon re-broadcast it, indicating that they are one hop from the basestation; nodes that hear those messages in turn re-broadcast them, indicating that they are two hops from the basestation, and so on. This process of (re)broadcasting beacons occurs continuously, such that (as long as the network is connected) all nodes will eventually hear a beacon message. When a node hears a beacon message, it chooses a node from which it heard the message to be its *parent*, sending messages through that parent when it needs to transmit data to the basestation[1].

This basic programming and data collection model has proven to be useful in a number of data collection applications, but as discussed in the introduction, is also somewhat limited. It only works with static networks, only allows applications that rely on a fairly limited subset of SQL to be expressed, and does not address how these sensor network applications integrate into a larger information processing ecosystem. We discuss how we have addressed these limitations in our recent work in the remainder of this paper.

3 Declarative Sensor Networks

Sensor networks are notoriously difficult to program. One reason for the success of sensor databases is the attraction of a familiar declarative language like TinySQL that hides the details of the network and its devices. In many cases this can radically simplify programs, replacing hundreds of lines of distributed, embedded C code with a few lines of SQL. But the ambition of this approach – to hide the sensornet entirely and focus on the data – also limits the utility of sensor databases as an aid to programmers. First, sensor databases address simple data acquisition and analysis, and this is only one piece of the sensornet programming puzzle. Second, like traditional database systems, the first generation of sensor databases were monolithic applications that could not be easily adapted for reuse in unanticipated settings. Unfortunately, experimental systems that cannot be reused in unintended ways tend not to be reused much at all.

In considering a second generation of database-style sensornet technology, it is worth revisiting the core programming challenges in this context. By definition, wireless sensor network programmers need to focus on three main issues: (a) power management (wireless), (b) data management (sensor), and (c) network design (networks). Our hypothesis in follow-on work is that a declarative approach can radically simplify (a) and (c), in addition to (b). Moreover, by handling all three in a uniform declarative language, opportunities emerge for cross-layer optimizations, and cross-disciplinary research impact [21].

To date, we have focused on networking issues, in the context of the *DSN* Declarative Sensor Network system [5]. This follows on the heels of our *P2* system for declarative overlay networks on the Internet [15]. Both systems take declarative network specifications in Datalog-like langauges, and compile them down to executable distributed dataflow programs. Our initial results in this direction have been quite positive. In typical examples in both the Internet and wireless domains, our declarative protocol implementations are orders of magnitude shorter to express than the traditional imperative programs, very close to the protocol pseudocode in syntax, and competitive in performance and robustness. On the sensornet front, we have implemented several very different classes of traditional sensor network protocols, services and applications entirely declaratively; these include radio link estimation, tree and geographic routing, data collection, and version coherency [5]. All

[1]In general, parent selection is quite complicated, as a node may hear beacons from several candidate parents. Early papers by Woo and Culler [22] and DeCouto et al [6] provide details.

```
% Initial facts for a tree rooted at "root"
dest(@AnyNode, root).
shortestCost(@AnyNode, root, infinity)

% 1 hop neighbors to root (base case)
R1 path(@Source,Dest,Dest,Cost) :? dest(@Source,Dest?,
        link(@Source,Dest,Cost).

% N hop neighbors to root (recursive case)
R2 path(@Source,Dest,Neighbor,Cost) :? dest(@Source,Dest?,
        link(@Source,Neighbor,Cost1),
        nextHop(@Neighbor,Dest,NeighborsParent,Cost2),
        Cost=Cost1+Cost2, Source!=NeighborsParent.

% Consider only path with minimum cost
R3 shortestCost(@Source,Dest,<MIN,Cost>) :?
        path(@Source,Dest,Neighbor,Cost),
        shortestCost(@Source,Dest,Cost2?, Cost<Cost2.

% Select next hop parent in tree
R4 nextHop(@Source,Dest,Parent,Cost) :?
        shortestCost(@Source,Dest,Cost),
        path(@Source,Dest,Parent,Cost?.
```

Figure 1: Tree routing in SNLog

.

have consisted of a small number of rules, which compile down to code that runs on the resource-constrained Berkeley Mote platform running TinyOS.

To illustrate declarative networking, we review an example from [5] that specifies shortest-path routing trees (as in Section 2) from a set of nodes to a set of one or more tree roots. In Figure 3 we present this protocol in DSN's SNLog [15]. Ignoring the "@" and "∼" symbols in Figure 3 for a moment, this looks much like a standard recursive Datalog program involving transitive closure. The two "fact" statements at the beginning of the program instantiate a tuple in each of the `dest` and `shortestCost` relations at each node in the network, stating that the destination is a node called "root", and (in the absence of other information) the distance between each node and the root is inifinity. The next two rules `R1` and `R2` find all possible paths from sources to tree roots. The last two rules prune this set: `R3` finds the min-cost path grouped by distinct source/destination pairs, and `R4` sets the `parent` of each source in each tree to be the `NextHop` in the shortest path. Finally, the query specifies that all such `parents` should be returned as output.

There are a few obvious distinctions from Datalog in Figure 3. First, each relation has one field prepended with the "@" symbol; this field is called the *location specifier* of the relation. The location specifier specifies data distribution: each tuple is to be stored at the address in its location specifier field. For example, the `path` relation is partitioned by the first (source) field; each partition corresponds to the networking notion of a local routing table. Second, the tilde ("∼") is a hint to the execution engine that the arrival of new tuples from the associated body predicate can be postponed, and need not trigger immediate reevaluation of the rule. More subtleties that distinguish SNLog from Datalog are discussed in [5].

Note that the `link` relation that captures radio link quality estimation has not been specified in the figure. As discussed in the initial DSN paper [5], it can be implemented as a "built-in" functional relation that calls a low-level radio driver in the OS, or it can be implemented in a fully declarative fashion as well using radio broadcast primitives. Loo et al. [15] show how location specifiers and link relations enable a compiler to generate a distributed protocol that is guaranteed to be executable over the underlying network topology captured by the link relation.

While this example may seem relatively simplistic, many of the other features we have written are almost as short, but can express networking issues from the lowest level of managing radio link quality, to very high-level application semantics such as object replica dissemination (the Trickle protocol mentioned above [14]). In all the examples, our code is orders of magnitude shorter to express than the native TinyOS implementations we replace [5], leading us to believe that a declarative approach can result in marked gains in software productivity and maintainability.

Given this starting point, we are pursuing several research issues that build on the basic language:

Query Optimization: Given that we can implement nearly all layers of a sensor network program in a declarative language, there is an opportunity to optimize across them all simultaneously. In addition to the intrinsic interest of this as a database research problem, it can lead to auto-generation of new network protocols, which can inform the development of networking technology. As an aspect of this challenge, we need an extensible optimizer to support new and custom optimization rules as they arise. To that end, we are designing a declarative *meta-optimizer* for P2 and DSN: an optimizer for distributed variants of Datalog that is itself written in Datalog. This turns out to be fairly natural: cost-based statistics-gathering can be expressed in queries, dynamic programming is a simple recursive program in Network Datalog, and recursive rule rewrites like Magic Sets Optimization are easy to express as well.

Heterogeneous Sensornets: The first generation of sensornet database systems focused on a network of homogeneous devices like Berkeley motes. A more realistic platform might mix these low-function devices with a variety of more capable but more power-hungry nodes. The design of protocols and algorithms that account for this heterogeneity is an important challenge in sensornets. Given P2 and DSN as execution engines targeted at large- and small-footprint devices, we are considering how to inform a query optimizer to synthesize protocols that make use of the heterogeneous resources intelligently.

Power Management: Related to the previous two points, a key aspect of sensornet programming is power management. Query optimization for sensor network protocols and applications must center on power as a key metric. Beyond that, the programming metaphor should expose power metrics and controls to the language, so that the programmer can control the optimization tradeoffs in an application-specific way. This requires a much richer interface between language and optimizer than the "optimizer hints" used in database systems. We are optimistic that a meta-optimizer framework can accelerate the exploration of these kinds of designs.

Distributed Statistical Methods: Sensor data is characteristically noisy and incomplete, so even the "data management" aspect of sensornet databases needs to be upgraded in the next generation. There has been a fair bit of research in recent years on statistical techniques to deal with these challenges, including work by the authors. But to date there has been no software infrastructure to make the programming of distributed statistical methods tractable. We hypothesize that languages like SNLog are actually a good fit to expressing distributed versions of these algorithms as well, and we have prototyped core statistical algorithms like Bayesian Belief Propagation. Making such programs fit into embedded sensor devices is an interesting additional challenge, given the memory footprint needed for maintaining sufficient statistics for these algorithms.

4 Complex Data Processing

Early tiny databases only allowed a limited set of basic database operations over data, which made them insufficient for a number of scientific applications that require sophisticated signal processing and analysis. For example, in one deployment in which we have been heavily involved, we are looking at the use of a collection of microphone-equipped sensors to localize, track, and classify wildlife in the field. One group of scientists we have been working with are particularly interested in populations of yellow-bellied marmots — large rodents endemic to the western US (see Figure 2). Such acoustic applications require high data rate (10's of kHz per channel) audio signals to be converted into the frequency domain (using an FFT operator) to identify frequencies that are characteristic of the animals being tracked. When such frequencies are detected, beamforming algorithms that perform triangulation of the signal are applied for tracking purposes.

Figure 2: An angry marmot.

To support these kinds of applications, we have developed a new programming language called WaveScript and a sensor network runtime system called WaveScope [9]. WaveScript is a functional programming language that allows users to compose sensor network programs that perform complex signal and data processing operations on sensor data. Similar to most streaming database systems, WaveScript programs can be thought of as a chain of operators that sensor data is pushed through; each operator filters or transforms data tuples and (possibly) passes those tuples on to downstream operators.

Unlike TinyDB, WaveScope doesn't impose a particular (tree based) communication model on sensors, but instead allows programs to operate on named data streams, some of which may come from remote nodes. To simplify programming tasks that involve data from many nodes, programmers can group nodes together, creating a single, named stream that represents the union or aggregate of many nodes' data streams.

Like TinyDB, WaveScope derives a number of benefits from the use of a high level programming model. Programmers do not need to worry about time synchronization, power management, or the details of networking protocols – they simply express their data processing application using the WaveScriptlanguage and the WaveScoperuntime takes care of executing these programs in efficient manner.

In the remainder of this section, we briefly overview the WaveScope data and programming model, focusing on expressing a few simple example applications.

4.1 The WaveScript Data Model

The WaveScript data model is designed to *efficiently* support high volumes of isochronous sensor data. Data is represented as streams of tuples in which each tuple in a particular stream is drawn from the same *schema*. Each field in a tuple is either a primitive type (*e.g.*, integer, float, character, string), an array, a set, a tagged union, or a special object kind of object called a *signal segment* (SigSeg).

A SigSeg represents a window into a signal (time series) of fixed bit-width values that are regularly spaced in time (isochronous). Hence, a typical signal in WaveScope is a stream of tuples, where each tuple contains a SigSeg object representing a fixed sized window on that signal. A SigSeg object is conceptually similar to an array in that it provides methods to get values of elements in the portion of the signal it contains and determine its overall length.

Although values within a SigSeg are isochronous, the data stream itself may be asynchronous, in the sense that the arrival times of tuples are *not* constrained to arrive regularly in time.

The WaveScope data model treats SigSegs as first-class entities that are transmitted in streams. This is unlike other streaming database systems [19, 3, 4] that impose windows on individual tuples as a part of the execution of individual operators. By making SigSegs first-class entities, windowing can be done once for a chain of operators, and logical windows passed between operators, rather than being defined by the each operator in the data flow, greatly increasing efficiency for high data rate applications.

4.2 The WaveScript Language

With WaveScript, developers use a single language to write all aspects of stream processing applications, including queries, subquery-constructors, custom operators and functions. In contrast, tiny databases and stream processing systems [19, 3, 4] typically provide a high level SQL-like language for writing queries but require user-defined functions to be written in external programming languages such as C. The WaveScript approach avoids mediating between the main script and user-defined functions defined in an external language, while further allowing *type-safe* construction of queries. In contrast, while SQL is frequently embedded into other languages, there is no compile-time guarantee that such queries are well-formed.

An example WaveScript script: Figures 3 and 4 show an WaveScript script query, first as a workflow diagram and then as the equivalent WaveScript subquery. The `marmot` function uses `detect`, a reusable detection algorithm, to identify the portions of the stream most likely to contain marmot calls, and then extracts those segments and passes them to the rest of the workflow, which enhances and classifies the calls. Several streams are defined: `Ch0..3` are streams of `SigSeg<int16>`, while `control` is a stream of `<bool,time,time>` tuples. Type annotations (*e.g.*, line 2) may be included for clarity, but they are optional. Types are inferred from variable usage using standard techniques [18]: for example, the definition of the `beamform` function implies that the type of `beam` is `Stream<float[360],SigSeg<float>>`.

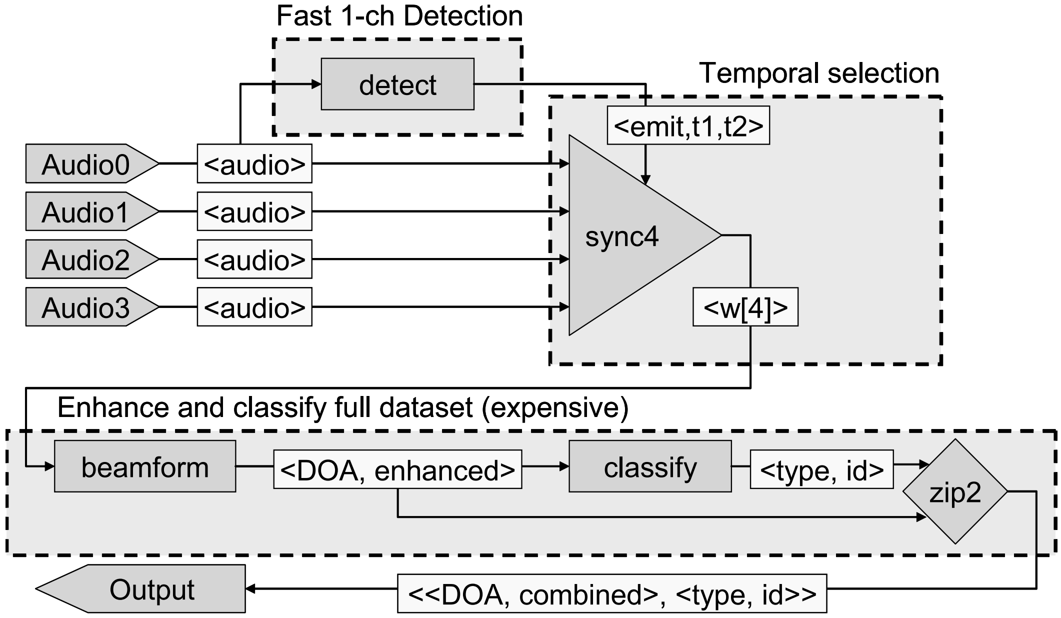


Figure 3: Marmot call detection workflow.

The `detect` subquery constructor has several parameters, including `marmotScore`, a custom function which computes the energy in frequency bands that are characteristic of marmot calls. This produces a `<bool,time,time>` stream of local marmot call detections that is merged with elements from the `net` input stream (representing a stream of tuples from a remote node). This merged and filtered stream is fed as a control stream to `sync4`, along with the four raw input streams from the audio sensors. `Sync4` aligns the four data streams in time, and "snapshots" synchronized segments of data according to the time-ranges specified in the control stream. In this way, `sync4` reduces the volume of input data by passing through only those segments of audio that `detect` suspects contain marmot calls.

Next, the synchronized windows of data from all four audio channels are processed by a beamforming algorithm. The algorithm computes a direction-of-arrival (DOA) probability distribution and enhances the input

```
fun marmot(Ch0, Ch1, Ch2, Ch3, net) {
  Ch0 : stream<SigSeg<int16>>
  // Detector on sensor inputs
  control : stream<bool,time,time>
  control = detect(Ch0, marmotScore, <64,192>,
                   <16.0, 0.999, 40, 2400, 48000>);
  // Control stream used to extract data windows.
  windows : stream<SigSeg<int16>[4]>
  windows =
    sync4(filter_lapped(merge(control, net)),
          Ch0, Ch1, Ch2, Ch4);
  // ... and process them: enhance
  beam<doa,enhanced> = beamform(windows, geometry);
  // ... and classify
  marmots = classify(beam.enhanced, marmotSig);
  // Return tuple of control and result streams
  return <control, zip2(beam, marmots)>;
}
```

Figure 4: Equivalent WaveScript subquery.

signal by combining phase-shifted versions of the four channels according to the most likely direction of arrival. The `beam` function returns a stream of two-tuples. We use a special binding syntax, "`beam<doa,enhanced> = ...`", to give temporary names to the fields of these tuples. That is, `beam.doa` projects a stream of direction-of-arrivals, and `beam.enhanced` contains the enhanced versions of the raw input data. Finally, this enhanced signal is fed into an algorithm that classifies the calls by type (male, female or juvenile), and, when possible, identifies individuals.

4.3 The WaveScope Runtime

In this section, we briefly review the runtime system which executes compiled WaveScript programs. Operators in compiled WaveScript query plans are run by a scheduler, which picks boxes one at a time and runs them to produce outputs.

A special *timebase manager* is responsible for managing timing information corresponding to signal data. This is a common problem in signal processing applications, since signal processing operators typically process vectors of samples with sequence numbers, leaving the application developer to determine how to interpret those samples temporally. A timebase is a dynamic data structure that represents and maintains a mapping between sample sequence numbers and time units. Examples of timebases include those based on abstract units (such as seconds, hertz, and meters), as well as timebases based on real-world clocks, such as a CPU counter or sample number.

Finally, a *memory manager* is responsible for creation, storage, and management of memory, particularly as is associated with SigSeg objects that represent the majority of signal data in WaveScope. Designing and efficient memory manager is of critical importance, since memory allocation can consume significant processing time during query processing.

These features – the timebase manager, memory manager, and scheduler – simplify the task of the programmer (who no longer has to worry about these details), and allow WaveScope to provide excellent performance. In our initial benchmarks of the marmot application, the current WaveScopesystem processes 7 million samples per second on a 3 GHz PC, and about 80K samples per second on a 400 MHz ARM platform (which has a 10x penalty for floating point emulation in addition to a reduced processor speed).

5 HiFi: A Data-Centric Architecture for Large-Scale Sensor Networks

Early work on wireless sensor networks, RFID, and other physical sensing infrastructure was necessarily focused on small-scale deployments designed in isolation from other computing and application systems. Starting from the assumption that these research efforts will ultimately lead to large-scale deployments, the HiFi project looks towards the future, addressing questions that must be answered before such technology can be effectively integrated and utilized in realistic big-science and global enterprise scenarios.

The core issues that arise involve systems architecture, programming paradigms, optimization, and integration with existing IT infrastructure. Furthermore, "macroscope" sensor network deployments such as those being proposed by emerging scientific collaborations represent costly infrastructure investments whose economics dictate that they be shared by many user communities and applications. Such multi-purpose deployments require a rethinking of fundamental sensornet service abstractions.

To meet these challenges, the HiFi project is developing abstractions and corresponding software infrastructure to support "high fan-in" architectures: widely distributed systems whose edges contain large numbers of receptors such as sensor networks, RFID readers, and network probes, and whose interior nodes are traditional host computers organized using the principles of cascading stream query processing and hierarchical aggregation. Such systems have a characteristic fan-in topology; Readings collected at the network edges are continually filtered, summarized, refined and passed towards the network interior. Application scenarios include RFID-enabled supply chain management, large-scale environmental monitoring, and various types of network and computing infrastructure monitoring. The central premise of HiFi is that High Fan-in systems should be programmable with a uniform, declarative, data-centric language.

The initial architectural work on HiFi was described in a paper that appeared in CIDR 2005 [8] and a prototype of the system was demonstrated at VLDB 2004. HiFi builds upon the TelegraphCQ stream query processing engine [4] for in-network processing, and the TinyDB system for pushing queries out to the network edge. The key additional consideration of this initial effort was the development of software components to efficiently execute distributed, heterogeneous sensing applications and to allow summary data and alerts to be rapidly propagated through the system while less time- critical detail data is efficiently archived for later search and bulk transmission.

Experience with this initial implementation proved the feasibility of using SQL-based processing across the entire network and raised several new challenges. These new challenges have been the focus of subsequent work on HiFi:

Virtual Device Interface: The declarative approach lends itself naturally to the development of a "Virtual Device" abstraction layer that shields application developers from the complexity of the underlying devices [12]. We developed an API for this abstraction that allows Virtual Devices to be constructed from multiple, heterogeneous physical devices, enabling them to provide higher quality, application-specific readings. Furthermore, we extended the API to support data cleaning functionally, thereby avoiding the need to include such functionality in every application [11].

Edge Event Processing: A clear lesson from our early implementations was the need to enable users to easily express event patterns of interest and to enable event processing to be pushed closer to the sensors themselves. We developed an event pattern extension to SQL and demonstrated the use of this functionality in an RFID-based library check-out system [20].

Shared Hierarchical Query Processing: Another key challenge was the development of algorithms for efficiently processing multiple queries in the hierarchical HiFi environment. This work involved the development of sophisticated query re-writes on window clauses and predicates in order to produce efficient shared plans [13].

The work described above represents a start towards the development of an architecture for large- scale sensor deployments but many problems remain to be solved. The key insight of the HiFi effort was that database techniques such as declarative query processing and various levels of data abstraction can and should play a central role in the development of such solutions. From this perspective, HiFi can be seen as a direct extension of the original TinyDB philosophy.

6 Supporting Mobility

Finally, existing tiny database systems assume a relatively static and well-connected network topology. Though networks are formed dynamically, and do adapt to failed nodes or small changes in connectivity, the basic tree-based routing protocols used in these systems do not work well when nodes are highly mobile or when the network is often disconnected.

In the CarTel [10] project, we are building a mobile sensor networking system for data collection from vehicles. Applications include collection of speed and position data for monitoring traffic conditions, monitoring of road-surface quality using accelerometers (to, for example, provide reports of potholes in the Boston area), and tracking of cellular and WiFi connectivity is over the entire city. By mounting sensors are cars it is possible to cover a very large urban area with just a few sensors (we cover thousands of miles of road a day using a collection of 27-cab mounted sensors.) CarTel includes a so-called *delay tolerant networking layer*, called *CafNet*. The idea with CafNet (and other systems, like DTN [7]) is that it buffers data packets during periods of disconnectivity and delivers those packets during fleeting moments of connectivity, for example, when two mobile nodes pass each other or when a mobile node comes into range of fixed networking infrastructure that can transmit data to a centralized collection point.

Delay tolerant networking by itself, however, is not sufficient to support mobility in sensor networks. The issue has to do with fact that in mobile settings, bandwidth is highly variable: during times of good connectivity, it may be abundant, allowing large amounts of data to be collected. During periods of poor connectivity, however, it may be so scarce that if applications continue to produce data at the highest rate possible, buffers will become huge. Because most networking stacks deliver data in a first-in/first-out fashion, this means that old data will often clog buffers, preventing new (or more important) data from being able to propagate out of the network.

For this reason, we have developed a new data management component for CarTel called ICEDB (for Intermittently Connected Embedded Data Base) that allows programmers to specify *prioritization policies* for data collected by their queries [24]. The basic query model is very similar to TinyDB: users pose continuous queries that request data be collected at a particular rate from mobile nodes (on cars, in this case.) Nodes buffer data until connectivity is available and then deliver it with CafNet. However, because connectivity may be fleeting or limited, when new data is produced by a query, it is added to a priority queue of data waiting to be sent by CafNet with a priority specified by an additional clause that can be added to every query.

ICEDB allows both inter- and intra-query priorities to be specified. Inter-query priorities specify that all of the results of a given query should be transferred before any of the results of some other query. Intra-query prioritization is more sophisticated. It allows each individual data item to be assigned a priority according to one of two user-defined prioritization functions. If a *local* function is specified, the priority of all data items waiting to be sent by a given query are re-evaluated using the local function whenever a new data item is produced. For example, if a car is sending a trajectory of <location, speed> pairs, simply sending the most recent positions may not be desirable if the application needs to accurately approximate the path taken by a car. Instead, it may be preferable to send points that are in the middle of the largest sections of unsent data. We call this policy “bisect”, as it recursively bisects the trajectory of points into smaller segments, providing an improved approximation of the entire path as more data is sent. To express this as a query, a user adds a “DELIVERY ORDER BY” clause to his query, as follows:

```
SELECT lat,long,speed
 FROM sensors
 SAMPLE PERIOD 1s
 DELIVERY ORDER BY bisect
```

Here, `bisect` is a built in function, but in general users can specify any function that takes as input a list of unsent points and totally orders them.

ICEDB also provides a *global* prioritization mechanism. Global prioritization is important in cases where there are multiple mobile sensors covering the same geographic area – for example, if there are several cars that have each driven on the same section of the freeway, the data from later cars may have lower priority if another car has already delivered similar data. Global prioritization works in conjunction with a server. The idea is that each mobile node produces a compact *summary* of the data it has available and sends that summary to the server before sending the (much larger) raw data. The server then assigns a priority to each element in the summary, and sends the new priorities back to the mobile nodes. The mobile nodes then order the raw (unsummarized) data according to the priorities assigned by the server. To specify a summary, users provide a SQL query that computes a summary over the buffer of readings waiting to be sent, as well as a function that runs on the server that assigns priorities to each element in the summary.

Taken together, these local and global prioritization schemes allow ICEDB to adapt to the variable bandwidth that is inevitable in highly mobile sensing applications. Without such adaptation, programmers would have little control of when their data is transmitted, forcing them either to artificially constrain data collection rates to work with minimal bandwidth, or requiring them to develop complex home-grown networking and prioritization solutions.

7 Conclusion

Early tiny database systems provided a powerful set of declarative tools for collecting data from static sensor networks. Building on these previous results, our recent research has shown how to extend these ideas to other domains, including as network protocol specification, mobile systems, high data rate and signal-oriented systems, and integrated systems that including traditional information processing infrastructure. These next-generation tools maintain much of the ease of use, simplicity, and optimizability offered by using a database-like approach to system design while addressing key limitations of early systems.

Acknowledgments

This work was supported by the National Science Foundation under grants CNS-0205445, CNS-0520032, and CNS-0509261, and by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan.

References

[1] D. Abadi and S. Madden. Reed: Robust, efficient filtering and event detection in sensor networks. In *VLDB*, 2005.

[2] B. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *Proceedings of the First Workshop on Information Processing in Sensor Networks (IPSN)*, April 2003.

[3] D. Carney, U. Centiemel, M. Cherniak, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, 2002.

[4] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of First Annual Conference on Innovative Database Research (CIDR)*, 2003.

[5] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *ACM SenSys*, 2007.

[6] D. S. J. D. Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Proceedings of ACM MOBICOM*, 2003.

[7] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the ACM SIGCOMM 2003*, August 2003.

[8] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design considerations for high fan-in systems: The hifi approach. In *CIDR*, pages 290–304, 2005.

[9] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden. The Case for a Signal-Oriented Data Stream Management System. In *Proc. CIDR*, Jan. 2007.

[10] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, E. Shih, H. Balakrishnan, and S. Madden. CarTel: A Distributed Mobile Sensor Computing System. In *Proc. ACM SenSys*, Nov. 2006.

[11] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom. Declarative support for sensor data cleaning. In *Proceedings of the ACM Conference on Pervasive Computing*, pages 83–100, 2006.

[12] S. R. Jeffery, M. J. Franklin, and M. N. Garofalakis. An adaptive middleware for supporting metaphysical data independence. In *VLDB Journal (to appear)*, 2008.

[13] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD Conference*, pages 623–634, 2006.

[14] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor network. In *NSDI*, 2004.

[15] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking with distributed recursive query processing. In *ACM SIGMOD International Conference on Management of Data*, June 2006.

[16] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of USENIX OSDI*, 2002.

[17] S. Madden, W. Hong, J. M. Hellerstein, and M. Franklin. TinyDB web page. `http://telegraph.cs.berkeley.edu/tinydb`.

[18] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 1978.

[19] R. Motwani, J. Widom, A. Arasu, B. Babcock, S.Babu, M. Data, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation and resource management in a data stream management system. In *Proceedings of First Annual Conference on Innovative Database Research (CIDR)*, 2003.

[20] S. Rizvi, S. R. Jeffery, S. Krishnamurthy, M. J. Franklin, N. Burkhart, A. Edakkunni, and L. Liang. Events on the edge. In *Proceedings of SIGMOD*, pages 885–887, 2005.

[21] A. Tavakoli, D. Chu, J. M. Hellerstein, P. Levis, and S. Shenker. A declarative sensornet architecture. In *WWSNA*, 2007.

[22] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of SenSys*, 2003.

[23] Y. Yao and J. Gehrke. Query processing in sensor networks. In *CIDR*, 2003.

[24] Y. Zhang, B. Hull, H. Balakrishnan, and S. Madden. Icedb: Intermittently connected continuous query processing. In *Proceedings of*, 2007.

Future Trends in Secure Chip Data Management

Nicolas Anciaux, Luc Bouganim and Philippe Pucheral
INRIA Rocquencourt and PRiSM Laboratory, University of Versailles, France
{firstname.lastname}@inria.fr

Abstract

Secure chips, e.g. present in smart cards, TPM, USB dongles are now ubiquitous in applications with strong security requirements. Secure chips host personal data that must be carefully managed and protected, thus requiring embedded data management techniques. However, secure chips have severe hardware constraints which make traditional database techniques irrelevant. We previously addressed the problem of scaling down database techniques for the smart card and proposed the design of a DBMS kernel called PicoDBMS. This paper summarizes the learning obtained during an extensive performance analysis of PicoDBMS. Then it studies to which extent secure chips hardware evolution should impact the design of embedded data management techniques. Finally, it draws research perspectives linked to a broader usage of secure chip data management techniques.

1 Introduction

Secure chips, i.e. chips with a high level of tamper resistance, are now ubiquitous in applications with strong security requirements. Secure chips are integrated in smart cards, Trusted Platform Modules (TPM [15]) and other forms of radio-frequency or pluggable smart tokens like USB dongles [7] or Mass Memory Cards [13]. Smart card is the most popular secure chip form factor and is used worldwide in applications such as banking, pay-TV, GSM subscriber identification, loyalty, healthcare and transportation. Now, large-scale governmental projects are pushing for an ever wider acceptance of smart cards (passport, driving license, e-voting, insurance or transport). Secure chips are also at the heart of computing systems, to protect PC platforms against piracy [15] or to enforce Digital Right Management in rendering devices [14]. Finally, chips are today integrated in a large diversity of usual objects to form an ambient intelligence surrounding. While security seems not the primary concern in this latter case, the strong demand of individual for enforcing their elementary rights to privacy quickly changes the situation.

In all these scenarios, data hosted by the secure chips is personal and must be carefully managed and protected. Embedded data management techniques are thus required to store the data securely (thanks to the chip tamper resistance) and act as a trusted doorkeeper, which authenticates each user (i.e., human being or software) and solely delivers her authorized view of the data. While a full fledged on board DBMS is not always required in all scenarios, access control policies may entail on-board computation of select, join and aggregate operators. Secure chips have however strong hardware constraints, and more importantly a very unique resource balance (e.g., powerful CPU vs. tiny RAM, very fast read vs. very low write in stable storage, etc) which make current database techniques, even those designed for lightweight DBMS [11] irrelevant.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

In a previous paper, we addressed the problem of scaling down database techniques for the smart card and proposed the design of a DBMS kernel called PicoDBMS [12]. We recently conduct extensive performance measurements of PicoDBMS on a Gemalto's smart card experimental platform and on a hardware cycle-accurate simulator allowing conducting tests on databases exceeding the smart card storage capacity. The objective was to assess PicoDBMS overall performance as well as the relative performance of candidate storage and indexing data structures.

The first contribution of this paper is to expose the learning obtained during this long and tricky experimentation process thus answering the question "an the PicoDBMS design effectively accommodate real hardware platforms?". Since the initial PicoDBMS design, secure chip hardware evolved following roughly Moore's law. A second question is then "Is scaling down database techniques for secure chips still a topical issue today?". The second contribution of this paper is to study the hardware progress and show how it impacts the design of data management techniques, opening interesting research issues. Finally, "Is PicoDBMS the definite answer to a sporadic problem or rather a first step towards a broader and longer term research?" is a legitimate question putting database management on secured chip in perspective. If PicoDBMS has proved to be an effective answer to the initial problem statement (i.e., on-chip DBMS), it does not answer all the situations where data management could benefit from secured chips. To illustrate this, chip manufacturers are announcing Mass Storage Devices [7, 10, 13] combining a secured chip with a huge amount (several GB) of unsecured external stable memory, dictating revisiting again the database technology on chip. The third contribution of the paper is to devise new research perspectives considering extending the sphere of confidentiality of secure chips toward external (unsecured) resources.

The paper is organized in three sections (Sections 2-4) addressing the three contributions mentioned above, followed by a conclusion (Section 5).

2 PicoDBMS design and evaluation

Current powerful smart cards include in a monolithic chip, a 32 bit RISC processor clocked at about 50 MHz, memory modules composed of about 96 KB of ROM, 4 KB of static RAM and 64 KB of EEPROM, and security modules enforcing physical security. The ROM is used to store the operating system, fixed data and standard routines. The RAM is used as working memory (heap and stack). EEPROM is used as stable memory to store persistent information, and holds data and downloaded programs (in case of multi-application cards).

The smart card internal resource balance is very unique. Indeed, the on-board processing power, calibrated to sustain cryptographic computations and enforce smart cards security properties, is oversized against the small amount of embedded data. In addition, EEPROM shares strong commonality with RAM in terms of granularity (direct access to any memory word) and read performance (60-100 ns/word), but suffers from a dramatically slow write time (about 10 ms per word). Finally, only a few bytes of RAM remain available for the embedded applications, the major part of the RAM being preempted by the operating system and the virtual machine (in case of Java enabled cards).

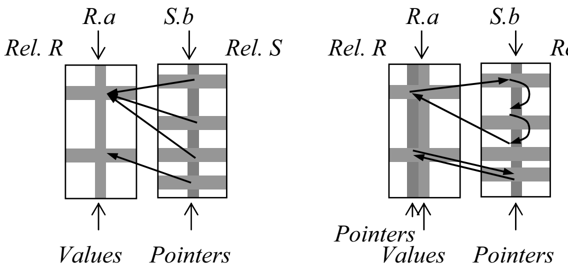
As mentioned above, PicoDBMS must act as a trusted doorkeeper delivering to each authenticated user authorized view of the data. A trivial solution would be to materialize each view in a separate file. However, this badly adapts to data and access control policies evolutions and would result in a huge data replication among files, thereby hurting the smart card limited storage constraint. Thus, a dynamic evaluation of the authorized views must be considered, leading to embed on chip the ability to compute database operators and run query execution plans implementing views. The database components that must be embedded are thus: a storage manager to organize data and indices within the chip stable memory, an access right manager to enforce grants and revokes on database views, a query manager to process execution plans and a transaction manager to enforce the ACID properties. Other database functions (e.g., query parsing and result sorting) do not impact data confidentiality and can be executed off card.

To match the smart card aforementioned hardware constraints, we have defined a set of design rules for on-chip database components [12]: *Compactness rule* (minimize data, index and code footprint), *RAM rule* (minimize RAM consumption), *Write rule* (minimize writes to very slow EEPROM), *Read rule* (take advantage of very fast reads in EEPROM), *Access rule* (take advantage of low granularity and direct reads in EEPROM), *CPU rule* (take advantage of the over-dimensioned CPU) and *Security rule* (never externalize private data, minimize code complexity). For the sake of conciseness, we recall below the storage, indexation and query execution techniques compliant with those rules and summarize performance results from [3]. The complete design of PicoDBMS is described in [12].

2.1 Storage and indexing models

The simplest way to organize data is *Flat Storage* (FS), where tuples are stored sequentially and attribute values are embedded in tuples. The main advantage of FS is access locality. However, FS is space consuming (all duplicate attribute values are stored) and inefficient (without index structures, FS relies on sequential scans for all operations). Since locality is no longer an issue in our context (Read and Access rules), pointer-based storage models may help combining indexing and compactness. The basic idea is to preclude any duplicate value to occur. Values are grouped in domains (sets of unique values) and attribute values are replaced by pointers within tuples (named tuple-to-value pointers). We call this model *Domain Storage* (DS). Obviously, attributes with no duplicates (e.g., keys) are not stored using DS but with FS. While tuple update and deletion are more complex than their FS counterpart, their implementation is more efficient in a smart card because the amount of data to be written is smaller (Write rule).

Let us now consider index compactness. A select index is typically made of a collection of values and a collection of pointers linking each value to all tuples sharing it (named value-to-tuple pointers). The collection of values can be saved since it exactly corresponds to a domain extension. To get the collection of value-to-tuple pointers almost for free, we store these pointers in place of the tuple-to-value pointers within the tuples. This yields an index structure that makes a ring from the domain values to the tuples. The ring index can also be used to access the domain values from the tuples and thus serves as data storage model. Thus we call *Ring Storage* (RS) the storage of a domain-based attribute indexed by a ring. The index storage cost is reduced to its lowest bound (one pointer per domain value), whatever the cardinality of the indexed relation, but slows down access to tuples attributes (project operation) since retrieving the value for the attributes means traversing in average half of the ring (i.e., up to reach the domain value).



a. Domain Storage. b. Ring Storage.

Figure 1: Storage models for foreign keys.

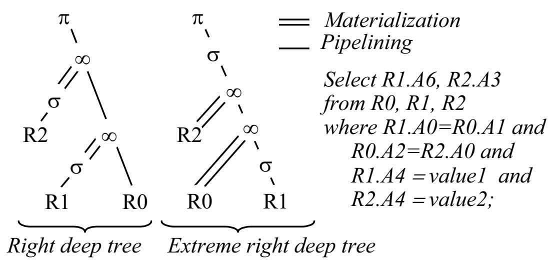


Figure 2: Extreme right deep tree example.

Join indices can be treated in a similar way. A join predicate of the form (*R.a*=*S.b*) assumes that *R.a* and *S.b* vary on the same domain. Storing both R.a and S.b by means of rings leads to define a join index. As most joins are performed on key attributes, R.a being a primary key and S.b being the foreign key referencing R.a, key attributes are stored with FS in our model. Nevertheless, the extension of R.a precisely forms a domain, even if not stored outside of R, and attribute S.b varies on this domain. Thus, DS naturally implements for free a unidirectional join index from S.b to R.a (Fig. 1a). Traversals from R.a to S.b can be optimized too by RS, a bi-directional join index being obtained by a ring index on S.b (Fig. 1b).

2.2 Query processing

Devising a query execution engine compliant with the design rules introduced above is challenging. Indeed, traditional query processing strives to exploit large main memory for storing temporary data structures (e.g., hash tables) and intermediate results and resorts to materialization on disk in case of memory overflow, which hurts both Read and Write rules. To address this issue, we propose query processing techniques that do not use any working RAM area (except for a small collection of cursors) nor incur any writes in stable memory.

Let us consider the execution of SPJ (Select-Project-Join) queries. PicoDBMS combines all operators in an extreme right-deep tree execution plan, which leads to pure pipeline execution without materialization. Fig. 2 compares a classical plan with an extreme right deep tree on a SPJ query. As left operands are always base relations, they are already materialized in stable memory. Pipeline execution can be easily achieved using the well-known Iterator Model. A query execution plan is activated starting at the root of the operator tree. The dataflow is demand-driven: a child operator passes a tuple onto its parent node in response to a next call from the parent. The Select operator tests each incoming tuple against the selection predicate. Depending on the storage model, attribute values are directly read in the tuple (FS) and/or reached by dereferencing a pointer (DS) and/or by following a pointers ring (RS). With RS, the selection predicate (or part of it whether multi-attribute) can be evaluated on the distinct values of a domain, and the matching tuples are directly retrieved following the relevant pointers rings. The project operator is pushed up to the tree since no materialization occurs. Project simply builds a result tuple by copying the value (FS) and/or dereferencing the cursors (DS) and/or following the pointers ring (RS) to reach the value present in the input tuple. With FS, joins are implemented by nested loops between the left and right inputs, since no other join technique can be applied without ad-hoc structures (e.g., hash tables) and/or working area (e.g., sorting). In case of indices (DS/RS), the cost of joins depends on the way indices are traversed. Consider the join between R (n tuples) and S (m tuples), S referencing R through a foreign key. With DS, the join cost is proportional to m starting with S (i.e., right input is S) and to $n \times m$ starting with R. With RS, the join cost becomes proportional to $n + m$ starting with R and to $m^2/2n$ starting with S (retrieving the R tuples associated to each S tuple incurs traversing half of a ring in average).

Let us finally consider the execution of the aggregate operator (sort is not described since it can be performed off card). At first glance, pipeline execution is not compatible with aggregation, typically performed on materialized intermediate results. Our solution exploits two properties: (i) aggregate can be done in pipeline if the incoming tuples are yet grouped by distinct values and (ii) pipeline operators are order-preserving since they consume (and produce) tuples in their arrival order. Thus, enforcing an adequate consumption order at the leaf of the execution tree allows pipelined aggregation. If the grouping attribute is not part of the right leaf relation, the execution tree has to be rearranged. Multi-attribute aggregation is trickier to implement since it imposes one (resp. several) Cartesian product operator at the leaf of the tree in order to produce tuples ordered by a couple (resp. tuple) of distinct grouping values.

2.3 PicoDBMS performance

Two different platforms, provided by our industrial partner Gemalto, have been used to measure PicoDBMS performance: a real smart card prototype compliant with the aforementioned hardware characteristics and a cycle-accurate hardware simulator of this prototype. The hardware simulator allows considering datasets exceeding the smart card storage capacity, up to 1MB. It delivers the exact number of CPU cycles between two breakpoints set in the embedded code and thus provides exact performance predictions.

Several versions of PicoDBMS have been measured: *P-FS* (PicoDBMS-FS version) supports only the FS storage model; *P-DS* is enhanced with the DS model for redundant attributes and foreign keys serving as a unidirectional join index as shown in Fig. 1a; *P-RS* also supports the RS model both for selection and bidirectional join indices. We evaluate the performance on a synthetic dataset composed of 5 joining tables, varying the number of embedded tuples (up to 50000 tuples), with simple (SP: select-project), regular (SPJ: select-project-join) and complex queries (SPJG: select-project-join-aggregate). We refer the reader to [3] for further details on the performance evaluation process and summarize below the main results regarding storage consumption, insertion and query performance.

Stable storage consumption depends both on the database and the DBMS code footprint. While code footprint reduction has been exploited as a marketing advantage by lightweight DBMS vendors, our experience shows that the challenge is more on reducing the database footprint than the code footprint. This is exemplified by the small difference between the footprints of *P-FS* (26 KB) and *P-RS* (39 KB). The database footprint is then the dominant factor except for very small datasets. As expected, *P-FS* is the least compact since it does not benefit from any form of compression; *P-DS* is the most compact, domains acting as a dictionary compression scheme (78% saving in our dataset); and the extra storage consumption incurred by *P-RS* is rather small (13% worse than *P-DS*), highlighting the high compactness of ring indices.

Insertion throughput is acceptable whatever the DBMS version. Insert performance is essentially impacted by the amount of writes in EEPROM. To this respect, *P-DS* performs better than *P-RS* itself performing better than *P-FS* roughly in the proportions mentioned above for the database footprint. Note that the tuple creation cost and the logging cost are independent of the relation size while integrity checking and domain update depend on the relation and domain cardinality, respectively. This balance the cost of updating indexing structures with the benefit provided to check integrity constraints.

Query execution performance, measured in throughput (result tuples produced per second), depends on the query complexity and on the presence of indexing structures. Simple queries (SP) reach high throughput delivery for any DBMS version (thousands of tuples/s). *P-RS* uses rings as selection indices, thus offering the highest throughput for queries with high to medium selectivity but looses its advantage on *P-FS* and k for queries with very low selectivity or no selection (projection overhead). Regular queries (SPJ) need *P-DS* or k to reach a good performance (hundreds of tuples/s). Indeed, *P-FS* yields low throughput because joins are performed by nested loops. With *P-DS*, the DS model implements a unidirectional join index imposing a unique join ordering in the query execution plan (e.g., from R to S on Fig. 1a) and precluding applying some selections first. *P-RS* takes advantage of the bidirectional join index provided by RS (see Fig. 1b), thus performing almost one order of magnitude better than *P-DS*. Complex queries (SPJG) require *P-RS* to fairly support aggregation (tens to a hundred of tuples/s). The important difference between SPJ and SPJG queries is explained by the high difficulty to handle aggregations without consuming RAM. Thanks to rings, joins within aggregation queries are handled with a limited degradation. The throughput remains stable when increasing the database size for aggregations on a single attribute. For multi-attribute aggregations, the ring index can help for only one of them, the others requiring Cartesian products, making the performance dependent on the database size.

The conclusion of this performance analysis is threefold. First, the performance results show without ambiguity that the PicoDBMS design (more precisely *P-RS*) effectively accommodates real hardware platforms thus answering positively the first question stated in the introduction. Second, the comparison between the PicoDBMS versions provides hints to select the most appropriate storage and indexing model for a given application. *P-FS* may suit very simple applications (small database, simple queries) if code simplicity is a main requirement. Note however that this intuitive choice should be reconsidered if database footprint or insertion cost is an important concern. *P-DS* sustains regular queries on large datasets, and *P-RS* fulfils the requirements of more complex applications involving aggregate queries. Third, this study highlighted two fundamental differences between traditional DBMS and on chip DBMS evaluation: (i) while performance is the usual metrics for traditional DBMS, no single metrics dominates when evaluating on chip DBMS (e.g., performance, code simplicity, storage consumption); (ii) the determination of on chip DBMS performance limits (e.g., maximal number of on-board tuples satisfying storage constraints or query performance requirements) is crucial and may help fixing co-design hints, considering that the hardware platform is not adjustable after manufacturing. We are currently working on a benchmark for on-chip DBMS taking these specificities into account.

3 Research Perspectives Driven by Hardware Properties

To answer the second question raised in the introduction, that is whether scaling down database techniques for secure chips is still a topical issue, we analyze the main foreseeable hardware trends for secure chips an discuss how they impact the design of embedded data management techniques.

3.1 Secure chips hardware trends

CPU resource: during the last ten years, embedded processors improved from the first 8-bit generation clocked at 2 MHz to the current 32 bit generation clocked at 50 MHz with an added cryptographic coprocessor. At least two factors advocate for a continuous growth of CPU power. First, the rapid evolution of secure chip communication throughput (e.g., high delivery contactless cards, USB cards) allows secure chip applications to evolve towards CPU intensive data flow processing using cryptographic capabilities (e.g., DRM). Second, many efforts from secure chip manufacturers focus on multi-applications and multi-threaded operating systems demanding even more CPU power. In addition, note that increasing the CPU power as little impact on the silicon die size, an important parameter in terms of tamper-resistance and production cost on large scale markets.

RAM resource: secure chips hold today a few KB of static RAM, almost entirely consumed by the operating system (and the Java virtual machine in Java enabled chips). The gap between the RAM left available to the applications on one side and the CPU and stable storage resources on the other side will certainly keep on increasing. First, manufacturers tend to reduce the hardware resources to their minimum to save production costs. The relative cell size of static RAM (16 times less compact than ROM and FLASH, 4 times less compact than EEPROM) makes it a critical component, which leads to calibrate the RAM to its minimum [2]. Second, minimizing the die size increases the tamper-resistance of the chip, thus making physical attacks trickier and more costly. RAM competing with stable memory on the same die, secure chip manufacturers favor the latter against the former to increase the chip storage capacity, thus enlarging their application scope.

Stable storage resource: secure chips rely on a well established and slightly out-of-date hardware technology (0.35 micron) to minimize production costs. However, the market pressure generated by emerging applications leads to a rapid increase of the storage capacity. Taking advantage of a 0.18 micron technology allows doubling the storage capacity of EEPROM memories. At the same time, manufacturers are integrating denser memory on chip, like NOR and NAND FLASH. NOR FLASH is well suited for code due to its "execute-in-place" property but its extremely high update cost makes it poorly adapted to data storage. NAND FLASH is a better candidate for data storage though its usage is hard. Reads and writes are made at a page granularity and any rewrite must be preceded by the erasure of a complete block (holding commonly 64 pages). In a long term perspective, researchers in memory technologies aim at developing highly compact non-volatile memories providing fast read/write operations with fine grain access (e.g., MEMS, PCM, etc.). But integrating these future technologies in a secured chip introduces intrinsic difficulties: very high security level that needs to be proved for any new technology, low manufacturing cost motivating the usage of old amortized technologies, complexity to integrate all components in the same silicon die.

To conclude, PicoDBMS has been designed a couple of years ago to tackle the rather unusual computing environment provided by secure chips. This environment was characterised by the three following properties: (P1) High processing power wrt the amount of RAM and on-chip data; (P2) Tiny RAM wrt the amount of on-chip data; (P3) Fast reads but slow writes in stable storage. As discussed above, these properties are expected to remain stable in the future, at least until the integration of emerging memory technologies like MEMS and

PCM. However, the diversification of hardware solutions (e.g., alternatives to EEPROM now exist) opens up new perspectives to tackle the secure chip constraints.

3.2 Hardware driven research issues

Stable storage technology: FLASH memory is becoming an effective alternative for secured chips due to its compactness, leading to new secure chip memory architectures mixing EEPROM, NAND and NOR FLASH. The different memory properties should be taken into account when designing embedded database components and specifically when designing the data storage and indexing models. [4] proposes techniques to store efficiently data on a smart card equipped with NOR FLASH but is limited to mono relation queries. NAND FLASH advocates for a sequential storage model (coarse grain erasure) while the reduced RAM of secured chips advocates for a highly indexed storage model. Hybrid chip architectures (e.g., mixing EEPROM and NAND FLASH) may lead to partition the data according to its update frequency/complexity (e.g., indices in EEPROM and data in FLASH). A co-design study would be particularly helpful to determine the best balance between the different types of memory. Long term memory alternatives could also be envisioned, including technologies like PCM, OUM and Millipedes. There are already some works on Millipedes [16] in a classical database context. Combining the memory properties with the other hardware constraints will undoubtedly generate important problems.

RAM: RAM is a crucial resource for the evaluation of database queries. PicoDBMS has been designed to cope with this issue (RAM rule), relying on an intensive use of indices. While this solution is convenient for the targeted context, indices may be disqualified in update intensive contexts (e.g., sensed data) because they increase update cost (especially in FLASH memory), they make update atomicity more complex to implement and they competes with on-board data. In addition, throwing away indices may help reducing the database footprint and the code complexity in contexts where performance is not the major concern. Thus, we consider that designing RAM constrained query execution techniques will remain a hot topic in the future. In [2] we proposed a first solution to minimize the RAM consumption in the query execution, assuming no index. First, we designed a RAM lower bound query execution mode inducing several iterations on the data to compute the query result. Second, we proposed a new form of optimization, called iteration filter, which drastically reduces the cost incurred by the preceding model, without hurting this RAM lower bound. Finally, we analyzed how to benefit from an incremental growth of the RAM amount. This work helps to determine: how much RAM should be added to reach a given response time, how much the expected response time should be relaxed to tackle a given query with a given quantity of RAM, how much data can be embedded in a given device without hurting an expected execution time. This is a first step towards more complete co-design work aiming at calibrating all resources of a hardware platform.

Cache conscious strategies: Another interesting research issue is improving the processor data cache usage when processing queries. Indeed, some announced smart cards are now endowed with a cache holding up to 1 KB. Since query execution with a constrained RAM induces numerous iterations on the data (e.g., to perform group by, joins without index), cache conscious algorithms could bring great performance improvements, diminishing the overheads of these iterations. Cache conscious strategies have been envisioned on traditional and main-memory DBMS and led to reorganize indices storage and data layout [1]. The challenge is that cache conscious processing generally leads to rethink data and index storage organization in a way which may contradict other concerns like data compactness and specific storage models dedicated to new stable memories. Co-design could again be helpful to calibrate the processor cache and the RAM, both competing on the same silicon die.

Contactless interactions: Contactless interfaces are more and more promoted by constructors and governmental organizations for their ease of use. While the current PicoDBMS design could adapt contactless smart cards, whether the processing requirements (in terms of query types and execution time) remain the same is an important question. For instance, severe response time constraints have to be enforced to avoid the card holder to stop when passing near a contactless reader. A possible solution could be to reduce the completeness/accuracy of the result for applications accepting partial results (e.g., top queries, approximate answers).

4 Extending the Sphere of Confidentiality to External Resources

Secured chips are often plugged into or linked to more powerful devices (e.g., cell phone, PDA, PC, and even servers). Thus, it does make sense to take advantage of the device resources to overcome the secured chip limitations. The idea is to extend the sphere of confidentiality provided by the security properties of the chip to external resources. This is actually the approach followed by the TPM [15] architecture where the secured chip protects the whole PC platform. This section sketches how to extend the sphere of confidentiality to external (unsecured) storage resources, to external processing resources and finally to a shared database server.

External Storage Resources: One of the main limiting factors of secure chips to address new domains of applications is the tiny capacity of the stable memory. To tackle this issue, smart card manufacturers are pushing new architectures combining in the same form factor (e.g., USB dongle) a smart-card-like secure microcontroller with an Gigabyte sized external (then unsecured) FLASH memory module (e.g., Gemalto's SUMO "smart card", Renesas X-Mobile Card [13]). To prevent from information disclosure and protect the integrity of the data stored on the unsecured external memory, cryptographic techniques (e.g., encryption, secure hash functions, etc.) must be employed. Some metadata (encryption keys, checksums, freshness information, bootstrap, etc.) and the query evaluator itself must remain embedded in the secured chip to deliver only the authorized data to the user. The problem is to combine cryptographic techniques and query execution techniques in a way satisfying three conflicting objectives: efficiency, high security and compliance with the chip hardware resources.

External Processing Resources: External computing resources (e.g., the CPU and RAM of the terminal the secure chip in connected with) could be exploited by delegating some expensive computation (e.g., sorting, duplicate removal). However, this incurs two complex problems. First, the correctness of the delegated computation must be checked, and if some techniques exist for simple selections [8], no satisfactory solution has been proposed so far for more complex operations. Second, delegating computation should not reveal any sensitive information. Since some information can be inferred even from encrypted data, this remains a challenging issue.

External Shared Database: Up to now, we considered that data sharing is under the control of a single chip, the external resources being considered as direct extensions of this chip. Assuming the database be stored on a remote and unsecured server and be shared among different users, the spectrum of applications grows. For instance, one may consider a shared database hosted by an — untrusted — Database Service Provider. The Database Service Provider Model has been studied for the first time in [9], but sharing was not a concern in this study. The sharing issue has been tackled in [5] thanks to an architecture called Chip-Secured Data Access (C-SDA). C-SDA is a client-based security solution where a smart card acts as an incorruptible mediator between a user and a remote encrypted database. The smart card checks the user's privileges, participates in the query evaluation and decrypts the final result before delivering it to the user. Since then, this hot topic generated several papers including [6]. So far, existing studies focused only on the confidentiality of the remote data but disregarded tamper-resistance (including opacity, integrity and freshness). Solutions relying on a secured co-processor at the server side could also be investigated and would raise new research challenges.

5 Conclusion

Secure chips have severe hardware constraints which have triggered a deep revisiting of traditional database techniques, with unusual design choices, leading to build a DBMS kernel called PicoDBMS. The objective of this paper was to answer three important questions. The first question was related to the feasibility of the PicoDBMS approach. We showed that the joint efforts from our team and from our industrial partner Gemalto led to a convincing prototype which effectively accommodates real hardware platforms. The second question was whether the problem addressed in the PicoDBMS study is still a topical issue despite the hardware progress. We analyzed the secure chips hardware trends and showed that many important research issues still need be explored. Finally, the third question was whether PicoDBMS opens up a broad and long term research. We showed that secured chip can be integrated in an insecure distributed computing system (e.g., a database system) and be the ultimate trusted party the complete security relies on. We sketched some research perspectives in this promising direction. Hence, we expect that this paper will contribute to the definition of an exciting research agenda for database management on secure chip.

References

[1] A. Ailamaki, D.J. DeWitt, M.D. Hill: Data Page Layouts for Relational Databases on Deep Memory Hierarchies. In Proc. of International Conference on Very Large Databases, 2002.

[2] N. Anciaux, L. Bouganim, P. Pucheral: Memory Requirements for Query Execution in Highly Constrained Devices. In Proc. of International Conference on Very Large Databases, 2003.

[3] N. Anciaux, L. Bouganim, P. Pucheral: Smart card DBMS: where are we now? INRIA technical report 80840, 2006.

[4] C. Bolchini, F. Salice, F. Schreiber, L. Tanca: Logical and Physical Design Issues for Smart Card Databases. *In Journal of ACM TOIS*, 21(3), 2003.

[5] L. Bouganim, P. Pucheral: Chip-Secured Data Access: Confidential Data on Untrusted Servers. In Proc. of International Conference on Very Large Databases, 2002.

[6] E. Damiani, S. De Capitani Vimercati, S. Jajodia, S. Paraboschi, P. Samarati: Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs. In Proc. of ACM Conference on Computer and Communications Security, 2003.

[7] Dekart SRL.: Dekart Smart Container, 2007. http://www.dekart.com/products/integrated/smart_container

[8] M. Gertz, A. Kwong, C. Martel, G. Nuckolls, P. Devanbu, S. Stubblebine: Databases that tell the Truth: Authentic Data Publication. *Bulletin of the Technical Committee on Data Engineering*, 2004.

[9] H. Hacigumus, B. Iyer, C. Li, S. Mehrotra: Executing SQL over Encrypted Data in the Database-Service-Provider Model. In Proc. of ACM International Conference on Management of Data, 2002.

[10] L. Hamid: New directions for removable USB mass storage, Press release, 2006. http://www.itwales.com/997893.htm.

[11] A. Nori: Mobile and embedded databases. In Proc. of ACM International Conference on Management of Data, 2007.

[12] P. Pucheral, L. Bouganim, P. Valduriez, C. Bobineau: PicoDBMS: Scaling down Database Techniques for the Smart card. *Very Large Data Bases Journal*, 10(2-3), 2001.

[13] Renesas Inc.: X Mobile Card, 2007. http://america.renesas.com/media/products/security/x-mobilecard/literature/X-MobileCardProductBrief.pdf

[14] The SmartRight Content Protection System. http://www.smartright.org. 2007.

[15] Trusted Computing Group. http://www.trustedcomputing.org. 2007.

[16] H. Yu, D. Agrawal, A. El Abbadi: Tabular Placement of Relational Data on MEMS-based Storage Devices. In Proc. of International Conference on Very Large Databases, 2003.

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903