International Journal of Information Technology and Business Management
29th July 2014. Vol. 27 No.1

ISSN 2304-0777 www.jitbm.com

LIGHTWEIGHT DATABASE SYSTEM (LWDBS): AN OVERVIEW

D.C. Igweze and E.O. Nwachukwu
Department of Computer Science, Faculty of Applied and Physical Sciences, University of Port Harcourt, Nigeria

dcigweze@yahoo.com

Abstract:

The continuous growth in the mobile devices and smart phones has continued to drive the need for lightweight databases. This paper presents an overview of lightweight database system, underpinning the theory behind lightweight database system (LWDBS). SQLite, XSet, and Amos II are some examples of lightweight database system discussed and their application areas highlighted.

Keywords: Lightweight, Database, SQLite, XSet, Amos II.

1 INTRODUCTION TO LIGHTWEIGHT DATABASE SYSTEM (LWDBS)

A lightweight database system (LWDBS) is a high-performance, application-specific Database Management system (DBMS). It differs from a general-purpose (heavyweight) DBMS in that it omits one or more features and specializes in the implementation of its features to maximize performance. Although heavyweight monolithic and extensible DBMS might be able to emulate LWDB capabilities, they cannot match LWDB performance [2].

General-purpose DBMSs are heavyweight; they are feature-laden systems that are designed to support the data management needs of a broad class of applications. Among the common features of DBMS are supports for databases larger than main memory, client-server architectures, checkpoints and recovery. A central theme in the history of DBMS development has been to add more features to enlarge the class of applications that can be addressed. As the number of supported features increased, there was sometimes a concomitant (and possibly substantial) reduction in performance. A hand-written application that does not use a DBMS might access data in main memory in tens of machine cycles; a comparable data access through a DBMS may take tens of thousands of machine cycles. It is well-known that there are many applications that, in principle, could use a database system, but are precluded from doing so by performance constraints (e.g., LEAPS).

DBMS CUSTOMIZATION: Extensible or open database systems were a major step toward DBMS customization e.g. TI's Open OODB, IBM's Starburst, Berkeley's Postgres/Miro/Illustra, Wisconsin's Exodus, and Texas's Genesis. Extensible DBMSs enabled individual features or groups of features to be added or removed from a general-purpose DBMS to produce a database system that more closely matched the needs of target applications. Unfortunately, extensible DBMSs were basically customizable heavyweight DBMSs; their architecture and implementations (e.g., layered designs, interpretive executions of queries) still imposed the onerous overheads of heavyweight DBMSs. While feature customization of DBMSs can indeed improve performance, it has been our experience that the gains are rarely sufficient to satisfy the requirements of performance-critical applications [3].

International Journal of Information Technology and Business Management
29th July 2014. Vol. 27 No.1

ISSN 2304-0777 www.jitbm.com

2 THE THEORY BEHIND LIGHTWEIGHT DATABASE SYSTEM (LWDBS)

Lightweight database management systems (LWDBS) appear to be the next major evolutionary trend in DBMS design. A lightweight DBMS is an application-specific, high-performance DBMS that omits one or more features of a heavyweight DBMS and specializes in the implementations of its features to maximize performance. Examples of LWDBs include main memory DBMSs (e.g. Smallbase, and Texas), primitive code libraries (e.g. Booch Components), Sqlite (sql-based architecture), Amos II and XSet (XML-based architecture). Each of these examples strips features from a general-purpose DBMS (e.g. Smallbase removes the databases larger than main memory feature, Texas removes client-server architectures, and the Booch Components further strip checkpoints and recovery) and demonstrates the performance advantages gained by doing so. In principle, an application achieves its best performance when it uses a "lean and mean" LWDB that exactly matches its needs.

There are broad application classes that require lightweight, not heavyweight, DBMSs. Because there are no formalizations, tools, or architectural support, LWDBs are hand-crafted monolithic systems that are expensive to build and tune. Clearly, what is needed are lightweight DBMSs that are extensible though they come with some problems.

Informally, a LWDB trades generality for increased performance. Table 2.1 is a partial list of DBMSs features whose absence offers opportunities for performance optimization. None of the features (or optimizations) listed are particularly novel. Rather, the choice of whether these and many other features should be omitted in LWDB construction arises all the time [3].

Table 2.1: Some DBMS Features And The Optimizations Possible If They Are Omitted.

Feature	**Optimization Possible if Feature is Omitted**
Concurrency Control. Multiple, simultaneous threads of control.	Avoid overhead of concurrency control.
Checkpoints and Recovery.	Use file copying to achieve database recovery and check-pointing.
OS Files. Use the file system and disk storage organizations provided by the operating system.	Read and write "raw" disks directly in order to optimize data placement on disk and buffer management.
Persistence. Store a copy of the database on disk.	Store data in transient memory and thereby avoid the over-head of disk I/O.
Large Databases. Databases larger than main memory.	Use regular memory addresses for tuples instead of general object identifiers (OIDs). Use main-memory storage structures instead of page-based disk structures.
Dynamic Queries.	Avoid query parsing, optimization, and interpretive execution at run-time.
Client/Server.	Run DBMS engine and client on the same processor and/or address space. Exchange data without extraneous copies, and implement operations via procedure calls (or inlining) rather than RPC.
Joins. Multi-relation queries.	Avoid overhead of multi-relation query optimization and maintaining relation statistics.
Data Distribution. Data resides on multiple computers connected by a communication network	Avoid overhead of network communication, fragmenting queries according to location, and maintenance of data location tables.
Set-Oriented Queries.	Lazily compute retrievals and avoid retrieving unnecessary tuples and storing large intermediate results.

Open source database giant such as MySQL and PostgreSQL are common knowledge; these free SQL database servers are commonly used by Web applications and other programs. For some uses, however, large database systems like Oracle, MySQL, PostgreSQL, etc are overkill.

International Journal of Information Technology and Business Management
29th July 2014. Vol. 27 No.1

ISSN 2304-0777 www.jitbm.com

A light weight database system is an embeddable database system that uses flat files. It does not need to be started, stopped, configured, or managed like other SQL databases. It is lightweight, simple, fast, and compact. And it works completely out of the box without any configuration [2].

When you quit most lightweight programs, there is no server hanging around eating up CPU and memory. To manipulate the database again, you simply reconnect. For large Web applications or programs involving multiple users that are making changes at the same time, lightweight database system may not be the best thing to use. But for simple, single-user, day-to-day use, a lightweight database system will most likely do what you need. Figure 2.1 is a diagram of lightweight database system indicating a database system without the client/server feature. This lightweight database system is optimized by eliminating the client/server database engine; thus, the system is light and there is no server to be configured, loaded or to be shutdown.

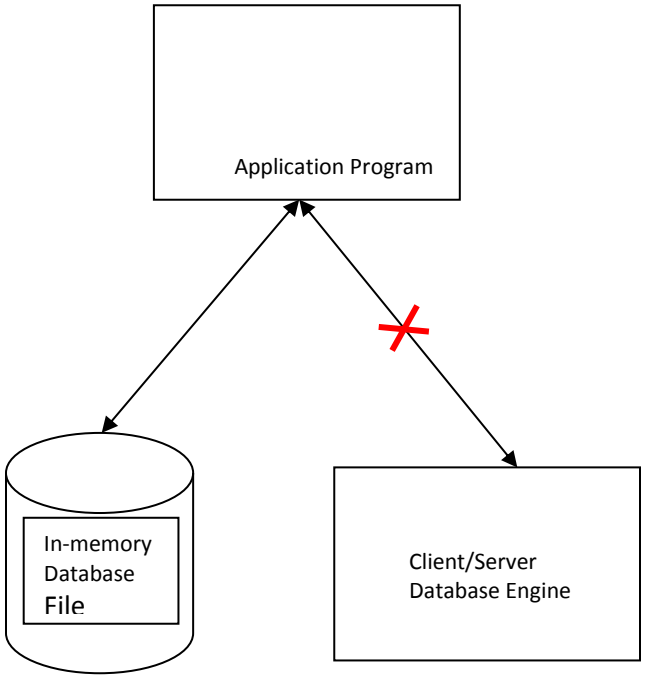

Fig. 2.1: Diagram Of A Lightweight Database System [2].

3 EXAMPLES OF LIGHTWEIGHT DATABASE

In this paper, SQlite is of major concern to us because it is a lightweight, embeddable, zero configuration, self-contained, main memory-based and server-less database with SQL-based architecture. However, XSet (XML-based architecture) and AMOS II are further discussed to reveal some lightweight features they possess.

3.1 SQLite As A Lightweight Database

SQLite is a software library that implements a self-contained, server-less, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world. The source code for SQLite is in the public domain.

SQLite is Self-Contained: SQLite is largely self-contained. It requires very minimal support from external libraries or from the operating system. This makes it well suited for use in embedded devices that lack the support infrastructure of a desktop computer. This also makes SQLite appropriate for use within applications that need to run without modification on a wide variety of computers of varying configurations. SQLite is written in ANSI-C and should be easily compiled by any standard C compiler. It makes minimal use of the standard C library. The required C library functions called are:

- memset()
- memcpy()
- memcmp()
- strcmp()
- malloc(), free(), and realloc()

SQLite can be configured at start-time to use a static buffer in place of calling malloc() for the memory it needs. The date and time SQL functions provided by SQLite require some additional C library support, but those functions can also be omitted from the build using compile-time options.

Communications between SQLite and the operating system and disk are mediated through an interchangeable VFS layer. VFS modules for Unix

ISSN 2304-0777 www.jitbm.com

and Windows are provided in the source tree. It is a simple matter to devise an alternative VFS for embedded devices.

For safe operation in multi-threaded environments, SQLite requires the use of mutexes. Appropriate mutex libraries are linked automatically for Win32 and POSIX platforms. For other systems, mutex primitives can be added at start-time using the sqlite3_config(SQLITE_CONFIG_MUTEX,...) interface. Mutexes are only required if SQLite is used by more than one thread at a time.

The SQLite source code is available as an "amalgamation" - a single large C source code file. Projects that want to include SQLite can do so simply by dropping this one source file (named "sqlite3.c") and its corresponding header ("sqlite3.h") into their source tree and compiling it together with the rest of the code. SQLite does not link against any external libraries (other than the C library, as described above) and does not require any special build support.

SQLite is Server-less: Most SQL database engines are implemented as a separate server process. Programs that want to access the database communicate with the server using some kind of interprocess communication (typically TCP/IP) to send requests to the server and to receive back results. SQLite does not work this way. With SQLite, the process that wants to access the database reads and writes directly from the database files on disk. There is no intermediary server process.

There are advantages and disadvantages to being server-less. The main advantage is that there is no separate server process to install, setup, configure, initialize, manage, and troubleshoot. This is one reason why SQLite is a "zero-configuration" database engine. Programs that use SQLite require no administrative support for setting up the database engine before they are run. Any program that is able to access the disk is able to use an SQLite database.

On the other hand, a database engine that uses a server can provide better protection from bugs in the client application - stray pointers in a client cannot corrupt memory on the server. And because a server is a single persistent process, it is able to control database access with more precision, allowing for finer grain locking and better concurrency.

Most SQL database engines are client/server based. Of those that are server-less, SQLite is the only one known to this author that allows multiple applications to access the same database at the same time.

SQLite is A Zero-Configuration Database: SQLite does not need to be "installed" before it is used. There is no "setup" procedure. There is no server process that needs to be started, stopped, or configured. There is no need for an administrator to create a new database instance or assign access permissions to users. SQLite uses no configuration files. Nothing needs to be done to tell the system that SQLite is running. No actions are required to recover after a system crash or power failure. There is nothing to troubleshoot. SQLite just works. Other database engines may run great once you get them going. But doing the initial installation and configuration can often be intimidating.

SQLite is Transactional: A transactional database is one in which all changes and queries appear to be Atomic, Consistent, Isolated, and Durable (ACID). SQLite implements serializable transactions that are atomic, consistent, isolated, and durable, even if the transaction is interrupted by a program crash, an operating system crash, or a power failure to the computer.

We here restate and amplify the previous sentence for emphasis: All changes within a single transaction in SQLite either occur completely or not at all, even if the act of writing the change out to the disk is interrupted by

- a program crash,
- an operating system crash, or
- a power failure.

The claim of the previous paragraph is extensively checked in the SQLite regression test suite using a special test harness that simulates the effects on a database file of operating system crashes and power failures.

Most Widely Deployed SQL Database: We believe that there are more copies of SQLite in use around the world than any other SQL database engine, and possibly all other SQL database engines combined. We cannot be certain of this since we have no way of measuring either the number of SQLite deployments or the number of deployments of other databases. But

ISSN 2304-0777 www.jitbm.com

we believe the claim is defensible. The belief that SQLite is the most widely deployed SQL database engine stems from its use as an embedded database. Other database engines, such as MySQL, PostgreSQL, or Oracle, are typically found one to a server. And usually a single server can serve multiple users. With SQLite, on the other hand, a single user will typically have exclusive use of multiple copies of SQLite. SQLite is used on servers, but it is also used on desktop PC, and in cellphones, and PDAs, and MP3-players, and set-top boxes. At the end of 2006, there were 100 million websites on the internet. Let us use that number as a proxy for the number of deployed SQL database engines other than SQLite. Not every website runs an SQL database engine and not every SQL database engine runs a website. Larger websites run multiple database engines. But the vast majority of smaller websites (the long tail) share a database engine with several other websites, if they use a database engine at all. And many large SQL database installations have nothing to do with websites. So using the number of websites as a surrogate for the number of operational SQL database engines is a crude approximation, but it is the best we have so we will go with it.

Now let's consider where SQLite is used:

- 300 million copies of Mozilla Firefox.
- 20 million Mac computers, each of which contains multiple copies of SQLite
- 20 million websites run PHP which has SQLite built in. We have no way of estimating what fractions of those sites actively use SQLite, but we think it is a significant fraction.
- 450 million registered Skype users.
- 20 million Symbian smartphones shipped in Q3 2007; newer versions of the SymbianOS have SQLite built in. It is unclear exactly how many Symbian phones actually contain SQLite, so we will use a single quarter's sales as a lower bound.
- 10 million Solaris 10 installations, all of which require SQLite in order to boot.
- Millions and millions of copies of McAfee anti-virus software all use SQLite internally.
- Millions of iPhones use SQLite
- Millions and millions of other cellphones from manufactures other than Symbian and Apple use SQLite. This has not been publicly acknowledged by the manufacturers but it is known to the SQLite developers.
- There are perhaps millions of additional deployments of SQLite that the SQLite developers do not know about.

By these estimates, we see at least 500 million SQLite deployments and about 100 million deployments of other SQL database engines. These estimates are obviously very rough and may be off significantly. But there is a wide margin. So the SQLite developers think it is likely that SQLite is the most widely deployed SQL database engine in the world.

3.2 XSet As A Lightweight Database For Internet Applications

Internet-scale distributed applications (such as wide-area service and device discovery and location, user preference management, the Domain Name Service, and personalized web portal pages) impose interesting requirements on information storage, management, and retrieval. They maintain structured soft-state and pose numerous queries against that state. These “Query Enabled" applications typically require the implementation of a customized, proprietary query engine that is often not optimized for performance and is costly in resources. Alternatives include using traditional databases, which can hamper flexibility, extensibility, and performance, all of which are critical requirements of Internet-scale applications, or a directory-based protocol, such as the Lightweight Directory Access Protocol (LDAP) [5].

Directory protocols pose composability problems and impose a rigid structure on queries. XSet, uses the eXtensible Markup Language (XML) as a data storage language, along with a high performance, main memory-based database and search engine. Using XML allows applications to use dynamic, simple, flexible data schemes and to perform simpler, but faster queries. XSet is a Java-based, easy to use, main memory, hierarchically structured database with partial ACID properties. Experimental measurements

ISSN 2304-0777 www.jitbm.com

show that XSet performance is excellent: insertion time is a small constant value, and query time grows logarithmically with the dataset size.

- Furthermore, XSet significantly outperforms platform-specific LDAP servers and XML-based databases on the LDAP directory benchmark. XSet is available for download, both as a stand-alone application and as a component of the Ninja service infrastructure [1].

The development of modern distributed applications has led to several interesting information storage, management, and retrieval requirements. In particular, an increasing number of applications are providing novel functionality by incorporating a fast search / information retrieval component. This new class of "Query Enabled" applications often maintain a mix of structured soft-state and durable hard-state, and poses numerous queries against that state. Examples of such Query Enabled applications are service- and device- location and discovery protocols, such as DNS and LDAP, and applications which make use of simple and fast query functionality, such as personalized web portal pages, searchable XML-enabled email systems and personal location trackers. The problems with these applications are three-fold: their extensibility is often very limited due to predefined, rigid data schemas; they pay for query power and flexibility with added schema complexity; and many of them offer similar functionality with significantly different implementations, duplicating effort and functionality.

These classes of applications are unified by using the eXtensible Markup Language [5]. XML as a data storage language along with a memory-based database and search engine is what we call XSet. We will define a set of data semantics for these applications that maximizes performance and concurrency. Finally, we provide a simple benchmark for evaluating XML query engines, such as XSet.

We chose XML as a description language because it offers numerous benefits including structured extensibility, strong flexible data validation through Document Type Definitions (DTD), powerful expressiveness, and ease of use. XML accentuates structure by making explicit the inherent structure of the data, without imposing a rigid schema. Furthermore, XML tags allow direct reference to data fields, extending expressiveness. Finally, XML is text-based, and offers data encapsulation in a human readable form without high overhead. These properties and a standardization effort make XML a natural choice for our needs.

XML is also useful because it provides a semi-structured data model. The structure and organization of data is often a limiting factor in how it can be used by applications. Data with a fixed, well-defined structure, as in a relational database, allows static typing, consistency checking, performance optimizations and well-defined queries, but can be confining should the data or query model evolve. Free-form data supports all data types and query models, but nothing can be known about the data statically. XML provides many of the benefits of both extremes. Not only can one reason about (and validate) the data a priori, but the data is also flexible enough to adjust to new data and query models [5].

There are several Query Enabled applications that use XSet for data management. Some of these applications use XSet to improve performance, while others are new applications that are made possible by XSet. As a whole, they demonstrate how XSet gives apps fast flexible querying capabilities with minimal overhead.

Berkeley Secure Service Discovery Service: Consider an academic campus of the near future, where the majority of the population is networked, and access the local computing resources using portable wireless devices. Users would like to utilize context-aware applications to access a wide range of dynamic data. For instance, a visitor wants to specify and find resources in their immediate surroundings, such as their meeting contacts or video projectors. Similarly, people who enter a building become temporary services, and register their personal preferences and profiles.

Other applications such as group pagers can then query the XSet server to locate and reach users [5].

Using traditional databases to solve this would require a large number of static schemas, ranging from personal location profiles to printer specifications, in addition to constant updating of these schemas as the format of data evolves. The transactional support and consistency guarantees available would be underutilized. Furthermore, these overhead costs would be duplicated per administrative domain, possibly exacerbated by incompatible databases and schemas [1].

ISSN 2304-0777 www.jitbm.com

The Service Discovery Service (SDS) is a wide-area soft-state-based directory service application implemented using XSet's querying functionality. “Soft-state" is the notion that all data has a finite lifetime, and systems provide fault-tolerance by supporting periodic refreshment of data, rather than specialized recovery modes. The SDS does not support transactions across queries, and leverages soft-state data to manage update consistency. Current performance analysis shows that XSet query latency composes a small fraction of the SDS query cost, and given expected optimizations on security and data distribution, the SDS will scale to extremely large datasets and query loads.

Personal Activity Coordinator: Another example of an XSet application is the Personal Activity Coordinator (PAC), an application written as part of the ICEBERG application architecture which acts as an intelligent cache of the current location and activities of ICEBERG users. Other ICEBERG applications query the PAC in order to determine the ideal contact point for incoming communication. The current implementation of the PAC uses an internal XSet server to store location- and application-specific information and services application queries [5].

Automatic Path Creator: One of the key components of the Ninja service infrastructure is the Automatic Path Creator (APC), which constructs a dataflow path between multiple Ninja services to compose a larger service. The APC uses an XSet server to store information on known sub paths and known services, and queries against it as part of a graph search algorithm to generate the logical path composition. Here, data is short-lived, and the fast query times of XSet are crucial to constructing paths within a reasonable response time.

Personalized Web Portal: Another product developed using the XSet database is SmartPages, a customizable web-based information portal. Smartpages retrieves interesting news articles, stock quotes and other dynamic content, stores it inside an XSet database, and presents it upon request. Users specifying their interest using Smartpage queries, and the XML content is transformed using XSL for their specific end-device. XSet could be used to provide similar functionality for other types of customized web portals, such as MyYahoo, MyExcite, and MyNetscape. These portals use user preference databases, based upon LDAP, real, or custom databases, to generate customized content for individual users [5].

3.3 Amos II As A Lightweight Object-Oriented Database

Amos II is a lightweight, Object-Oriented (OO), multi-database system. Object-Oriented multi-database queries and views can be defined where external data sources of different kinds are translated through AMOS II and integrated through its OO mediation primitives. Through its multi-database facilities many distributed AMOS II systems can inter-operate. Since most data reside in the data sources, and to achieve good performance, the system is designed as a main-memory DBMS having a storage manager, query optimizer, transactions, client/server interface, etc. The AMOS II data manager is optimized for main-memory and is extensible so that new data types and query operators can be added or implemented in some external programming language. Such extensibility is essential for data integration [6].

The AMOS II system has its roots in the workstation version of the Iris system, WS-Iris, and the DAPLEX functional data model. The core of AMOS II is an object-oriented, open, lightweight, and extensible database management system (DBMS). To achieve good performance AMOS II is designed as a main-memory DBMS. Each AMOS II server is also a DBMS of its own containing all the traditional database facilities such as a storage manager, a recovery manager, a transaction manager, and an OO query language, named AMOSQL. The system can be used as a single-user database or as a multi-user server to applications and to other AMOS II systems. The data manager is designed for main-memory and is optimized for efficient execution when the entire database fits in main-memory [4].

AMOS II is distributed mediator system allowing a number of AMOS II mediator servers to communicate over the Internet. Applications can access data from several distributed data sources through a collection of mediators. The mediator servers have facilities for translating, combining, reconciling, and abstracting data through OO views over other mediators and external data sources. The abstraction services allow presenting different object view hierarchies in the different mediators. The mediator servers appear as virtual database servers having data abstractions and an OO query language.

ISSN 2304-0777 www.jitbm.com

AMOS II mediators are composable since a mediator server can regard other mediator servers as data sources. A single AMOS II server can also assume more than one role and serve more than one application simultaneously. Different interconnecting topologies can be used to connect mediator servers depending on the integration requirements of the environment. Some of the servers can be configured as translators, wrapping different kinds of data sources, e.g. access to relational databases through ODBC or access to XML files. We use the term translator (rather than the commonly used term wrapper) since translators are complete AMOS II systems which can wrap more than one data source and support semantic data abstractions and conversions from its data sources through OO views. A translator is thus, also a mediator that provides a virtual OO database server layer that transparently translates data from some data sources. Wrappers are usually simpler interfaces to data sources while a translator can contain several wrapper subsystems for different data sources. Mediator servers can also act as integrators which combine and convert data from other mediator servers through OO views. As for translators, these OO views provide a virtual OO database layer to be transparently accessed from clients and other mediator servers. Conflicts and overlaps between similar real-world entities being modelled differently in different data sources can be reconciled through the mediation primitives of AMOSQL. The declarative multi-database query language AMOSQL requires queries to be optimized before execution. The query compiler translates AMOSQL statements into object calculus and algebra expressions in an internal simple logic based language called ObjectLog, which is an OO dialect of Datalog. As part of the translation into object algebra programs, many optimizations are applied on AMOSQL expressions relying on their OO and multi-database properties. During the optimization steps, the object calculus expressions are re-written into equivalent but more efficient expressions. For distributed multi-database queries the query decomposer distributes each object calculus query into local queries to be executed in the different distributed AMOS II servers and data sources. A cost-based optimizer on each site translates the local queries into procedural execution plans in an OO algebra, based on statistical estimates of the cost to execute each generated query execution plan expressed in the OO algebra. A query interpreter finally interprets the optimized algebra to produce the (partial) result of a query [4].

The query optimizer is extensible through a generalized foreign function mechanism, multi-directional foreign functions. It gives transparent access from AMOSQL to special purpose data structures such as internal AMOS II meta-data representations or user defined storage structures. The mechanism allows the programmer to implement query language operators in an external language (Java, C or Lisp) and to associate costs and selectivity estimates with different user-defined access paths. The architecture relies on extensible optimization of such foreign function calls. They are important both for accessing external query processors and for integrating customized data representations from data sources. To achieve good performance we have carefully optimized the representation of critical system data structures, e.g. the storage manager, object representation, type information, and the representation of function definitions. We use tailored main memory data structure representations of system objects, rather than, e.g. storing them in relational tables represented as B-trees. For example, our object identifiers are represented as variable length records with pointers to data structures representing type-information, function definitions, dependent objects, etc. It is crucial that system information is represented efficiently, since it is extensively looked up during both compilation and interpretation of AMOSQL functions. The storage manager has an incremental garbage collector for removing unused data [6].

AMOS II runs under Windows NT. The system uses around 350KB of code and 1500KB of meta data. The system has client-server and inter-database communication primitives whereby AMOS II servers can communicate over TCP/IP. The multi-database architecture of AMOS II allows several AMOS II systems to connect and communicate over a network using TCP/IP. There are furthermore AMOSQL data interoperability primitives to exchange data between different AMOS II systems and to mediate semantically heterogeneous data. AMOS II systems can communicate and they can be configured in different modes with respect to how they interact with other systems. The system can be configured in two dimensions:

ISSN 2304-0777 www.jitbm.com

- It can be a single-user, a server or an embedded system, where a single-user AMOS II system is a private database; a server is servicing several other AMOS II systems; and an embedded system is linked to some application.
- It can be a stand-alone, or a mediator system; where a stand-alone system is an isolated database, and a mediator accesses data from some mediator(s) or data source (s) [6].

4. CONCLUSION

In conclusion, we have explained the buzz phrase: lightweight database, highlighting the relevance of lightweight database in the advancement of mobile phones technology. This paper has brought out the fundamental theory behind the development of lightweight database systems and some examples x-rayed. The problems associated with using a heavyweight database system in performance-specific applications is a perfect candidate for the application of lightweight database technology.

5. ACKNOWLEDGEMENT

I am deeply indebted to my supervisor, Prof. E. O. Nwachukwu of the Department of Computer Science, University of Port Harcourt, for his fatherly advice, visionary suggestions and encouragements while this research effort lasted. Also, I am grateful to Mr. Bartholomew Eke and the entire staff of Computer Science Department University of Port Harcourt, Nigeria, for providing all the necessary and enabling academic support for the success of this Paper.

Reference

[1] Agrawal, D., Abbadi, A. E., and Singh, A. K. (1993). Semantics-based Correctness Criteria for Databases. ACM Transactions on Database Systems 18, 3, 460-486.

[2] Hipp Richard (2006). An Introduction to SQlite: Google Tech Talks, May 31 2006. http://www.youtube.com/watch?v=giAMt8Tj-84 Retrieved in October 2013.

[3] Jeff Thomas and Don Batory (1995). P2: An Extensible Lightweight DBMS, UTCS Technical Report 95-04, Department of Computer Sciences, The University of Texas, Austin, Texas 78712.

[4] Orsborn K., and Risch T. (1996): Next Generation of O-O Database Techniques in Finite Element Analysis. Int'l. Conf. on Computational Structures Technology, Budapest, Hungary.

[5] Zhao B. Y and Joseph A. D (2000). XSet: A Lightweight Database for Internet Applications. Computer Science Division, University of California, Berkeley.

[6] Vanja J. and Tore R. (1999). Distributed Mediation Using a Lightweight OODBMS. Linkoping University, Sweden.