Database Programming

Dr. Dobb's Journal

MAY 2012

SQLite on Android

With a little care, SQLite can be used as a data store or full database on Android devices

Next

ALSO INSIDE

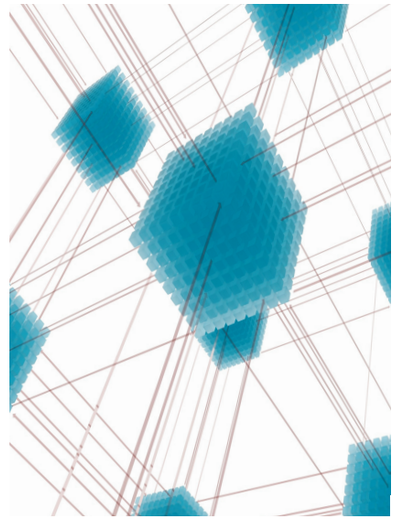
UBM TechWeb

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

May 2012



COVER STORY

More on DrDobbs.com

C Finally Gets A New Standard

Tom Plum explores the new additions to C in a series of articles, starting with language-level atomic operations and thread primitives.

http://www.drdobbs.com/cpp/232800444

Benchmarking Block-Swapping Algorithms

Algorithms for swapping blocks of data appear simple, but vary widely in their performance profiles. Choosing the right one can be crucial to good performance.

http://www.drdobbs.com/parallel/232900395

Auto Types and Range-Based *For* Statements in C++11

Two really handy features in C++11 are the *range-based for* statement and the auto type *specifier*. The former allows you to iterate over collections using a much more compact form of expression, and the latter takes some of the headache out of the complex type declarations encountered in the standard library.

http://www.drdobbs.com/blogs/cpp/232900460

PCs and Tablets: The Convergence Is Happening Now

The new iPad? Meh. I'm running Visual Studio 11 on Windows 8 (64-bit) on my tablet. Yes, my tablet.

http://www.drdobbs.com/mobile/232900072

Intel's 50+ Core MIC Architecture: HPC on a Card or Massive Co-Processor?

Will Intel's Knights Corner chips function as co-processors like GPUs, or will they be standalone many-core Linux systems? The two approaches present very different performance profiles.

http://www.drdobbs.com/parallel/232800139

Mailbag

More on CISC/RISC and Tasks as the fundamental unit of computing

RISC at the Heart of CISC

As part of the thread in the last two installments of the Mailbag regarding when exactly Intel moved to RISC processing inside its x86 processors, we received this additional insight:

"During 1992, I was a Technical Assistant for Dave House at Intel. At that time, he was the senior vice president for Corporate Strategy and we spent almost 50% of our time on the road talking to investors, students, and at conventions.

When he discussed the original Pentium 60, there was a slide on the screen behind him with a die shot of the Pentium. Each major function of the die was outlined and labeled. Of particular interest in this discussion was the block that converted CISC instructions to RISC code. The processor also had two pipelines: one integer only, the second floating point (though it could function in integer mode. Internally it would decide which pipe to use next — a kind of prediction scheme).

As Dave explained it, the whole issue of CISC or RISC became a non-issue at this point. A programmer could use whatever instructions were deemed most efficient. The lookup that converted CISC to RISC had been optimized to produce the most efficient RISC code for those instructions on that processor, so functionality of CISC wouldn't be improved by tweaking RISC code.

I might also point out that the 486-series processors were the last generation along the Intel roadmap in which it was possible to count clock cycles the way it had always been done in microcontrollers. Since the Pentium had two pipes with internal prediction switching and the CISC/RISC lookup, it might complete an instruction one or two clock cycles ahead of schedule."

— Robin McCain
Cinematiks, LLC

Tasks As The Fundamental Unit of Computing

Two editorials (http://is.gd/MU3P5f and http://is.gd/gr3suD) regarding the use of tasks as the fundamental unit of computing in the dawning world of concurrency drew very interesting observations.

"There's an interesting model of parallelism called pi-calculus (http://is.gd/UcRLop), also implemented as a Java extension (http://is.gd/lWTKvz), which demonstrates that you don't need to send tasks around (the work to be done), it's enough to send channels around. Additionally, communication is synchronous, unlike actors, so this can effectively replace synchronization primitives in most situations.

When you've been raised in thinking sequentially, it may take some time to express yourself idiomatically in this model. But experience shows that you quickly end up writing code that looks totally different than the idiomatic sequential version, and that this code is usually pretty intuitive, highly modular, and highly scalable. I would say, quite close to the spirit of the "task programming" model that you envision here."

— Patrick Viry
Paris, France

"I read your editorial today with much interest having arrived at a very similar point of view over the last few years.

I have several implementations in place designed and built around fine-grained tasks driven by a thread pool. One application, for example, distributes mine scheduling tasks across a WAN utilizing desktop and server idle time and has been in production for approximately four years; another is a peer-to-peer camera sharing application, another a package manager. To date, all have been built in C++ on Windows. The choice of C++ has been happenstance and I don't see any reason why a .NET implementation wouldn't work.

All the applications use I/O completion ports for task queuing. The general flow is that a task arrives in the I/O completion port queue and is picked up by the next available thread, which, based on information contained in a completion key and overlapped data structure, is routed to the appropriate task handler. The task consists of several steps in a sequence. When the sequence is complete, or yields, the current thread returns to the queue for more work. Tasks can be both externally generated (HTTP requests) or internally generated by currently running tasks.

Tasks encapsulate relatively fine-grained logical actions, for example, receive HTTP request, resolve request handler, post response, etc. The steps within a task are micro-grained, if you will. Each step is a function that performs a simple operation such as open a WinHttp session, create a file. Deep call chains are avoided as much as is practical. Each step returns to the dispatcher, which transfers control to the next step. A task can be visualized as a circular arrangement of steps around the dispatcher.

In addition to better matching the application to the processing capabilities of the machine, I have found other benefits. The dispatcher is a natural central point to attach additional smarts, such as instrumentation, exception handling, and diagnostics. Over the course of several implementations, the dispatcher has evolved into a small virtual machine with data structures for describing the tasks and their constituent steps. A useful side effect of this evolution has been the separation between the application protocol, that is, the set of logical tasks, vs. the implementation of the micro-grained steps. I've found the step routines tend to stabilize fairly quickly and remain so. Most of the change is at the protocol level, which seems to be appropriate. The protocol headers have also become a useful shared design and documentation artifact.

At present, the VM supports some simple flow-control constructs; `throw`, `catch`, and `finally`; and assertions. As it has evolved, it is an adjunct to native C++ code. Its focus has been on defining sequencing and flow control for native functions using native data structures. It is driven by a set of macros and the resulting logic is compiled directly into the executable. This is a useful capability, but there is a plan to also develop an external compiler with a cleaner syntax.

I've been fortunate to have customers willing to let me do as I see fit and have been able to drive the design and implementation. In other cases, I've had to work much harder to get the concept across. Even with some successful implementations as a basis for discussion, the approach wins few friends. As you allude to, the sense of profound unfamiliarity engendered by anything not sanctioned and supported by the chosen framework or language seems a fairly effective barrier to adoption. Being able to add another advocate such as you can only help."

— John Revill
Australia

Have a correction or a thoughtful opinion on *Dr. Dobb's* content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of *Dr. Dobb's*.

Microsoft's New Simple Database

Microsoft finally delivers a usable, no-admin, small-footprint database that can power real apps

by Dino Esposito

On the Windows platform, SQL Server can serve most any need for storing relational data. SQL Server is a top quality DBMS system and offers just about everything developers could want: CRUD services, transactionality, concurrency, programmability, and a large ecosystem of tools. As an enterprise-level application, SQL Server runs as a service. When run as a separate application, SQL Server may live on a physically distant machine and require a connection and data exchange. While this extra layer of complexity is acceptable, and often required, in enterprise applications, it sometimes proves to be too much for simpler scenarios such as storing relational data on mobile devices.

Mobile devices, however, are currently the most popular scenario where developers feel the need for something simpler and more manageable than a full installation of SQL Server. As a trainer and consultant, I often build quick prototypes of applications that need to read and write to a database. When I pass my code to students or customers, it would be much easier to just offer a database file. This is not something that can happen with the complex, service-based nature of SQL Server.

For years, Microsoft Access was used as a relational database for simple applications. However, Access was never designed to serve as a database engine outside single-user, local applications. When deployed as the data store of a Web application, it shows serious performance issues, the most dramatic of which is the bottleneck in writing. To offer something simpler than the full SQL Server, Microsoft introduced SQL Server Express. That product fixed all the issues of Access, and can definitely be used as the back-end engine in simple and low-volume applications. But one big issue remained — SQL Server Express doesn't support x-copy deployment well — which is precisely what made Access a common choice. In simple applications, often all that you want is the ability to copy a database on a server, and continue without having to deal with configuration.

Microsoft has now released SQL Server Compact Edition 4.0 (SQLCE), and this is the definitive response to the demand for having a simple and lightweight database for developers who need to do simple things. SQLCE supports databases as files. Both the SQLCE runtime and database files can be packaged and deployed with applications without relying on a database administrator. Compatible with .NET 3.5 and later, the runtime of SQLCE is in-process, multithreading, and multiuser. More importantly, it starts up with the hosting application, whether ASP.NET or Windows.

SQLCE database files are characterized by the SDF file extension and must be created and managed entirely in code. In other words, there's

no console application (such as SQL Server Management Studio) to help you organize multiple tables, indexes, and stored procedures. You do most of this work — specifically design, editing, and running queries — from the user interface of Visual Studio 2010 Service Pack 1. SQLCE is the default database for WebMatrix and it works in medium- or partial-trust environments, which makes it a good fit for simple and/or mobile scenarios.

As far as programming is concerned, SQLCE offers a T-SQL syntax. Although this is inferior to full SQL Server, it is powerful enough to cover some of the most-advanced common scenarios. For example, SQLCE supports paging queries and offers an API to retrieve metadata information out of the database. In addition, when used with Entity Framework, SQLCE supports server-generated columns (such as identity columns) and GUID rows. Finally, the database files can be encrypted using the SHA2 encryption algorithm. SQLCE has a query syntax and internal model of the database files similar to SQL Server, which makes moving from a solution based on SQLCE to one based on full SQL Server a smooth transition.

You can code against a SQLCE database using either low-level ADO.NET commands or the increasingly popular code-first approach. Code-first means that you create a set of entities and relationships as plain C# classes and then use a mix of conventions and code to map these classes to SQLCE database tables.

The latest version of SQLCE (version 4.0) is not compatible with Windows Phone. This simply means that you can't leverage the SQLCE engine and design tools of Visual Studio, but reading and writing relational data from within a Windows Phone application is still possible. Windows Phone 7.5 supports a LINQ-to-SQL API that can create a SQLCE database file in the isolated storage of the device and operate on it.

Overall, the best approach for mobile developers is not so different from what you do when using SQLCE, say, in a WebMatrix application. You create plain C# classes and use an attribute to mark them as tables and their properties as columns. Next, based on the information in the connection string, you create a new database programmatically or read an existing one — which can be an existing SDF file created with SQLCE in another application. Finally, you use LINQ-to-SQL to prepare queries, and the data context object to arrange writes and deletions.

In the end, even though the SQLCE runtime is not supported in Windows Phone, learning how to code against SQLCE database files will help when you do Windows Phone database development. In Windows Phone, while you can't rely on the SQLCE engine, you do have an API to read and write SQLCE files.

SQLCE is the real deal — the lightweight DBMS system that developers have been waiting for ever since the limitations of Access first became clear. As such, SQLCE may finally sound the death knell for the popular Microsoft Access.

— Dino Esposito is a frequent contributor to **Dr. Dobb's.** *His most recent book,* **Programming Microsoft ASP.NET MVC** *(http://is.gd/DTIw3U) was published by Microsoft Press.*

Comment

Persisting Data on Android Using SQLite

With a little care, SQLite can be used as a data store or full database on Android devices

By Gaddo F. Benedetti

As with most platforms, Android gives us a few options to store data so that it persists even after we've terminated the application. Of the various ways we can do this, text files — either stored in the application's own files directory or to the phone's SD card — represent one approach. Preferences are also frequently used to store data because they can be both hidden from the user and persist as long as the application is installed. And while not strictly speaking in the same category, Assets can be useful for storing read-only data, too. Assets are essentially files that you bundle into the application package prior to compilation in the project assets folder, which can be accessed at runtime. I will take a closer look at these later.

Sometimes, however, we need to be able to carry out complex operations on persistent data, or the volume of data requires a more efficient method of management than a flat text file or preference entry will allow. This is where a mobile database comes in.

Android comes with SQLite (version 3.5.9+), which has been available on the platform since release 1.5 of the OS (the "Cupcake" release). For readers unfamiliar with SQLite (http://sqlite.org/), it is a self-contained, server-less, transactional SQL database. While it has its limitations, SQLite serves as a powerful tool in the Android developer's arsenal.

What I'll principally cover here is one way to use a SQLite database in Android, concentrating on its management; specifically, creation and update, rather than all the runtime operations.

Managing SQLite

To begin with, we can manage SQLite using a class that extends `SQLiteOpenHelper`, which comes with a constructor and two required methods; `onCreate` and `onUpgrade`.

Naturally, the first of these is executed when the constructor is instantiated; it is here that, via the superclass, we provide four important pieces of data:

- Context. This is the context of the application. It can be useful to set this in the constructor and store it locally for later use in other methods.
- Database name. This is the filename (as a String) of the physical database file being accessed.
- Cursor factory. The cursor factory in use, if any.
- Database Version. This is the version of your database (as an integer), which I'll discuss in more detail later. Your initial version should be 1.

For our example, we put these four pieces together and get the following:

```
class DB extends SQLiteOpenHelper {

    final static int DB_VERSION = 1;
    final static String DB_NAME = "mydb.s3db";
    Context context;

    public DB(Context context) {
        super(context, DB_NAME, null, DB_VERSION);
        // Store the context for later use
        this.context = context;
    }
```

The constructor does two things. First, it checks whether the database exists and, if not, will call the `onCreate` method. Second, if the database does exist, it will check whether the existing database version number differs from the one implemented in the constructor, so as to determine if the database has been updated. If it has, the `onUpgrade` method will be called.

Additionally, as we now know that the `onCreate` method is called only when the database does not exist, it can be used as a handy way to determine if you're dealing with a first run of the application following installation. As such, you can use this method to call any other methods that you need executed only on the first run, such as EULA dialogs.

Let's look at the database itself. For the purposes of this article, I'm just going to use a very simple employee database with a SQL creation script as follows:

```
CREATE TABLE employees (
    _id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    ext TEXT NOT NULL,
    mob TEXT NOT NULL,
    age INTEGER NOT NULL DEFAULT '0'
);
```

We can easily construct this by hard-coding and executing the creation SQL, line by line, in our code as follows:

```
@Override
public void onCreate(SQLiteDatabase database) {
    database.execSQL(
        "CREATE TABLE employees ( _id INTEGER PRIMARY KEY "
    + "AUTOINCREMENT, name TEXT NOT NULL, ext TEXT NOT NULL, "
    + "mob TEXT NOT NULL, age INTEGER NOT NULL DEFAULT '0')");
}
```

As you can see, this can get quite unwieldy once the database reaches a certain size and complexity, so the ideal solution would be to bundle a SQL creation script as an asset file. To use this approach, you need to write a method that takes in a SQL script from the assets directory and parses it, executing it line by line:

```
@Override
public void onCreate(SQLiteDatabase database) {
    executeSQLScript(database, "create.sql");
}

private void executeSQLScript(SQLiteDatabase database,
                              String dbname) {
ByteArrayOutputStream outputStream = new
                              ByteArrayOutputStream();
    byte buf[] = new byte[1024];
    int len;
        AssetManager assetManager = context.getAssets();
    InputStream inputStream = null;

    try{
        inputStream = assetManager.open(dbname);
        while ((len = inputStream.read(buf)) != -1) {
            outputStream.write(buf, 0, len);
        }
        outputStream.close();
        inputStream.close();

        String[] createScript =
                  outputStream.toString().split(";");
        for (int i = 0; i < createScript.length; i++) {
               String sqlStatement = createScript[i].trim();
            // TODO You may want to parse out comments here
            if (sqlStatement.length() > 0) {
                  database.execSQL(sqlStatement + ";");
            }
        }
    } catch (IOException e){
        // TODO Handle Script Failed to Load
    } catch (SQLException e) {
        // TODO Handle Script Failed to Execute
    }
}
```

While this is a more complex approach than simply executing each SQL statement for very simple databases, it quickly pays divi-

dends once the database becomes more complex or you need to pre-populate it. You'll also see that I abstracted the creation into a separate method called `executeSQLScript` so that it can be reused in other situations, as I'll explain later on.

Interacting with the Database

Now that the database is created, I want to be able to interact with it. Here's a brief rundown:

The first step is to open the database and there are two ways to do this: using `getReadableDatabase()` and `getWritableDatabase()`. The former is faster and uses fewer resources. It should be used for anything that does not require writing or changing the database. The latter call is better suited to `INSERT`s, `UPDATE`s, and the like.

In Android, recordsets are returned from queries as `Cursor` objects. To carry out a query, use either the `query()` or `rawQuery()` method. For example, the following two calls return exactly the same data:

```
DB db = new DB(this);
SQLiteDatabase qdb = db.getReadableDatabase();
Cursor recordset1 =
     qdb.query("mytable", null, null, null, null, null, null);
Cursor recordset2 = qdb.rawQuery("SELECT * FROM mytable", null);
```

The first call uses a bewildering number of parameters. They are the table name, a string array of the column names, the `WHERE` clause, the selection arguments, the `GROUP BY` clause, the `HAVING` clause, and the `ORDER BY` clause, respectively. You'll note that setting many of these as `null` has the effect of their being treated as wildcards or omitted altogether.

Most of these parameters are fairly straightforward to anyone familiar with SQL. The selection arguments, however, require a little more explanation because they form a string array that interacts with the `WHERE` parameter, systematically replacing any "`?`" characters in the clause with a value from the array.

With the `rawQuery()` approach, there are only two parameters: the first is the SQL query, and the second is the selection argument — akin to those used in the query method. Selection arguments may be preferable to use with complex queries, such as those that use `JOIN`s.

Similarly, `INSERT`, `UPDATE`, `DELETE`, and a range of other common operations are handled with methods similar to `query()`; or, as with `rawQuery()`, they can be executed as raw SQL code using `execSQL()`.

Database Upgrades

Returning to management of the database, let's look at the tricky scenario of database upgrades. Over time, an app will likely change. New functionality may be added, or it may be better optimized. These changes, in turn, may lead to a requirement to upgrade the database schema and change the value for `DB_VERSION` in the updated application code to reflect this.

One potential problem is that replacing our database with a new version will end up invalidating the previous version and lead to the loss of data that was present in existing user installations. A second problem is that, once our application has reached a point whereby more than two versions have been released, we cannot presume that the user has been diligently upgrading all along, so a simple upgrade from one version to the next may no longer work.

To deal with this, we already know that if we introduce a new database version, the `onUpgrade()` method will be called. So ideally, we can use our SQL script parser method and execute one or more update scripts.

Let's look at what we intend to change in Version 2 of our database in the example:

- Normalize the phone number data (extension, mobile) into a separate "numbers" table, which includes a numeric field to denote the type of phone number.
- Add a salary field to the employee table.

Using the Version 1 schema as a starting point, this can be handled by writing a SQL script that updates the schema and then populates it with the data from the older version:

```
CREATE TABLE numbers (
   _id INTEGER PRIMARY KEY AUTOINCREMENT,
   employid INTEGER NOT NULL,
   number TEXT NOT NULL,
   ntype INTEGER NOT NULL DEFAULT '0'
);
CREATE INDEX employid ON numbers(employid);

INSERT INTO numbers (employid, number, ntype) SELECT _id, ext, 0
    FROM employees;
INSERT INTO numbers (employid, number, ntype) SELECT _id, mob, 1
    FROM employees;

CREATE TABLE temp (
     _id INTEGER PRIMARY KEY AUTOINCREMENT,
     name TEXT NOT NULL,
     salary INTEGER NOT NULL DEFAULT '0'
);
INSERT INTO temp (_id, name) SELECT _id, name FROM employees;

DROP TABLE employees;
ALTER TABLE temp RENAME TO employees;
```

Naturally, the more complex the changes in your database schema, the more complex the script you'll need to write to handle this. SQLite has more limited support for SQL than many databases, so sometimes you'll need to devise workarounds for these limitations. For example, in the aforementioned update script, I had to employ a temporary table as a workaround for the lack of `DROP COLUMN` functionality.

Now that I have the SQL upgrade script, I need to handle how it is executed when the `onUpgrade` method is called. One approach is to do the following:

```
@Override
public void onUpgrade(SQLiteDatabase db,
    int oldVersion, int newVersion)
{
    if (newVersion > oldVersion) {
        switch (oldVersion) {
            case 1:
                  executeSQLScript(database, "update_v2.sql");
            case 2:
                  executeSQLScript(database, "update_v3.sql");
        }
    }
}
```

There are two things to note in this code. The first is that I check to see whether the new database version is greater than the old one. I wrote the code this way because the `onUpgrade()` method will be called any time these two versions are different, leading to a situation in which version downgrades are also possible. Our code does not envisage this ever occurring, but an `else` clause and accompanying code could be added to handle this.

The second thing is that there are no `break` statements between the cases in our switch. This is because each script simply updates from one version to the next, meaning that an upgrade from Version 1 to 3 will first run the upgrade script from Version 1 to 2, then continue on to run the upgrade script from Version 2 to 3. If the database is already at Version 2, it will simply skip the first script and only run the upgrade script from Version 2 to 3.

Thus, each time you upgrade the database, you will only need to replace the create script with one that reflects the new schema (for new installs), and an update script (that handles only an update from the previous version) to handle all possible upgrades. Meanwhile, in our Java code, we need to update the value for `DB_VERSION` and, naturally, any operations that may be affected by the new database schema.

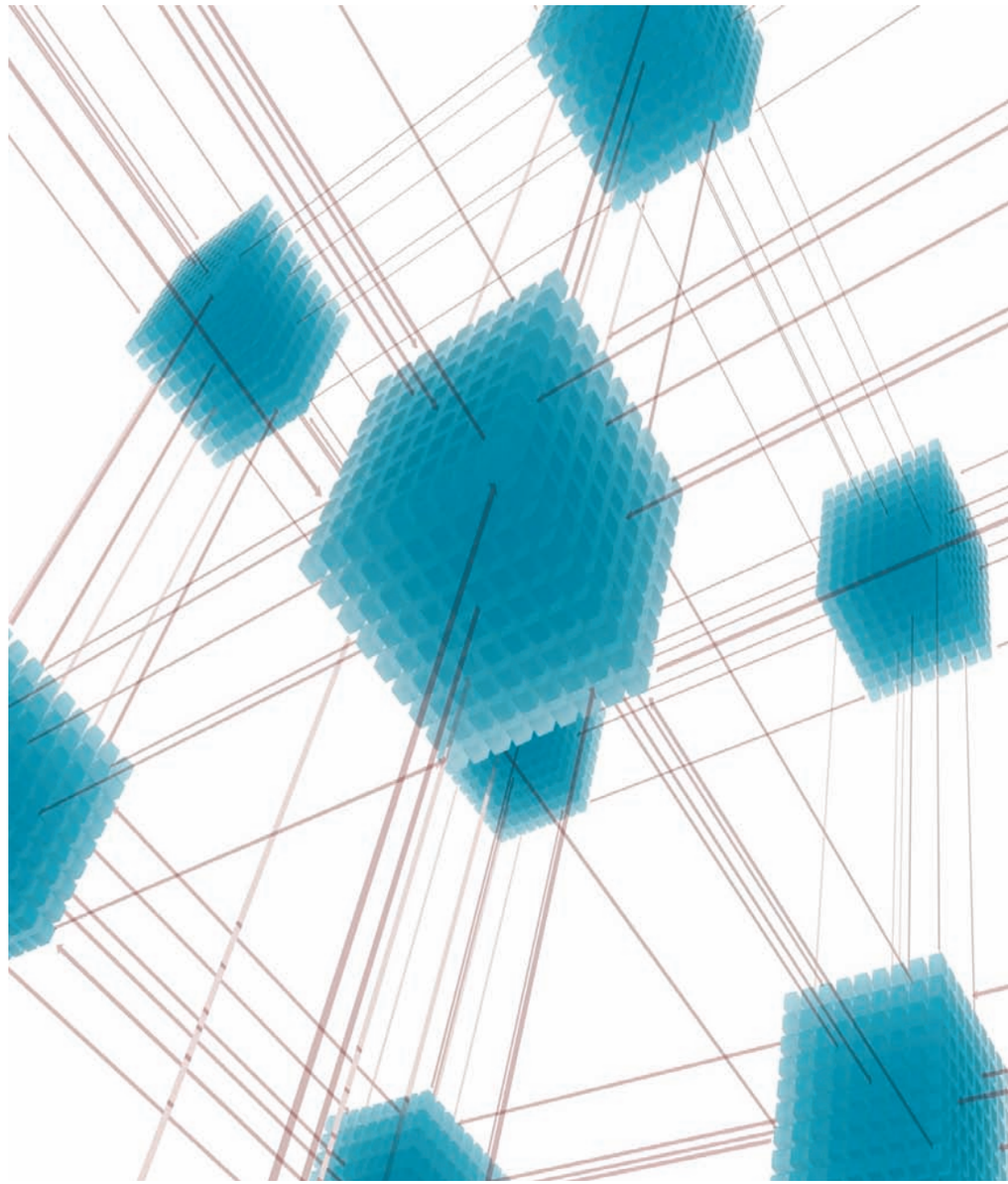
Conclusion

SQLite can be a very useful means to store and manage data in a persistent manner on Android devices. However, as with any database, care needs to be taken to administer it correctly, particularly with regard to version changes.

Using the script parser solution and saving this part of the application logic as a series of SQL scripts is an efficient and simple management technique to avoid having to write complex methods to the core application to handle each upgrade, thus allowing you to concentrate on the application's business logic instead.

— Gaddo F. Benedetti is presently an independent analyst-engineer who consults for various European clients in the telecom and Internet industries.

Comment



Kernels, Contexts, Threads, and Extensible Database Architecture

If you are building a new database management system or simply want to write better database applications, you should be aware of the extensible architecture of operating systems and database managers.

By Ken North

Servers with multiple processor cores, NewSQL, cloud computing, and advanced analytics have put a spotlight on extensible database architecture. It has long been possible to use a plugin to add functionality to a database, such as a new type, just as we can use add-ins to extend a Web browser. But today we see plugins gaining traction for a variety of purposes. If you are building a new database management system (DBMS) or simply want to write better database applications, you should be aware of the extensible architecture of operating systems and database managers.

To undertake a study of kernels, microkernels, and database architecture, a good departure point is found in some features of modern processors.

Processor Operating Modes

Processors have long supported multiple operating modes to segregate system services and application code at execution time. The purpose of these designs was to afford a level of protection for the routines that comprised the system software. The idea is that you don't want faulty application code to overwrite a part of memory that's used by the operating system (OS); overwriting memory or data can cause a crash.

The venerable IBM System 360 mainframe of the 1960s was a processor that offered a separation between System State and User State. The former was intended for code that executed to provide OS services when a program executed a supervisor call (SVC). The latter was how applications normally operated.

IBM OS/360 was designed during an era of expensive main memory, when machines with 512K of memory were the top of the line. To keep the operating system to a manageable size, IBM developed an extensible OS architecture. It supported the execution of Transient SVCs: small (2K) special-purpose modules that executed in a transient program area of memory.

The popular PDP 11 series minicomputer offered multiple processor modes (Supervisor, User, and Kernel mode); it also offered separate memory spaces for instructions and data. The Intel Xeon processor (in fact, all Intel processors since the 386) also has three processor modes. A program can operate in Real Mode, Protected Mode, or SMM Mode. These processor architectures enable us to write programs that change from one mode to another for critical operations, such as invoking an interrupt service routine that executes in kernel mode. A mode transition goes hand-in-hand with a context switch that preserves and restores the state of the CPU. Table 1 shows a breakdown of processor modes.

Memory-managed computers and multiple processor modes added the term "context switch" to the programming lexicon. In a protected memory environment, "context" refers to a virtual address space containing executable code. Various activities, such as invoking a kernel mode function or preempting a task, will initiate a context switch for the purpose of storing state information for the old executing task and retrieving state information for the new code.

Computer	Processor Modes (Operating States)
IBM System 360	System, User
Digital PDP 11	Supervisor, User, Kernel
Intel Xeon	Real, Protected, SMM

Table 1.

The context switch has been integral to modern systems and is essential for multitasking operations. With today's processors and operating systems, the term takes on a broader meaning. You can have register, task, thread, or process context switches. Some processors now provide hardware support for context switching by providing a data segment for task state information.

Evolution of Kernels

The kernel and microkernel form the code that's at the heart of an operating system. For processors with multiple operating modes or rings, the kernel or microkernel executes at the most privileged state (supervisor or kernel mode).

The microkernel was developed in part because monolithic OS kernels were becoming large and unwieldy, making them difficult to maintain. It seemed practical to implement support for new filesystems, protocol stacks, and device drivers in code that was outside the kernel and kernel address space. The microkernel enabled OS architects to develop privileged servers — such as networking servers, filesystem servers, and display servers — to deliver essential services without being built into the kernel. It's possible to start and stop these servers without rebooting, which makes it easier for development and testing. If a server crashes, it does not corrupt data in the kernel address space.

There have been divergent schools of thought about which is the preferred architecture, the microkernel or monolithic kernel. Dr. Andrew S. Tanenbaum (http://is.gd/SbLxn0) of MINIX fame favors the mi-

crokernel architecture, whereas Linus Torvalds stated his preference for the monolithic kernel architecture in a 2006 "Hybrid kernel, not NT" forum post (http://is.gd/UIW4Di).

Torvalds felt the separation of address spaces, making it impossible to share data structures, introduced too much complexity: *"Microkernels are much harder to write and maintain exactly because of this issue."*

The dichotomy of opinion among OS developers also extends to the hypervisor and virtualization community. Xen and VMWare are examples of the microkernel architecture, whereas MokaFive BareMetal is based on the monolithic model.

SQL Database Servers

Prior to the famous Tanenbaum-Torvalds debate (http://is.gd/LPS7iI), the SQL database community had already been weighing monolithic database architecture against client-server database processing. The latter moved database processing off shared, centralized mainframes to distributed, dedicated servers. DBMS architects also came to favor a separation between the database kernel and services, plug-ins, or extenders.

This was noticeable over time as we saw SQL platforms evolve in a Swiss Army Knife manner. Instead of adding replication, for example, to the core database engine, it made more sense to write a replication server.

When you install a high-end SQL product, it uses a layered architecture that brings up a number of servers at startup time. The layers of services in a DBMS can include storage management, network access, lock management, replication, security, query, cache management, memory management, OLAP services, and so on.

For a product such as Oracle Database, the kernel is the core of the server process. Network communications map to a layer known as the Transparent Network Substrate (TNS), which was designed to support heterogeneous connectivity. An Oracle server runs a TNS listener for processing database requests using client-server protocols. Listeners are linked to end points (port numbers) for HTTP, FTP, and XML DB requests. The listener forwards requests to either a shared server or a dedicated server process. The kernel uses background processes, such as the Process Monitor, System Monitor, Log Writer, and Database Writer. It also uses slave processes working in the background, such as I/O Slave Processes and Parallel Query Slaves.

One benefit of the Oracle kernel architecture is being able to provide a top-level trace. For debugging and tuning queries, it helps you see what's happening during execution of a query, from the kernel down to other processes.

Threads and Processes

The Oracle Database architecture for threads and processes differs depending on whether Oracle is running on a UNIX or Windows platform. On UNIX machines, Oracle Database uses a process to implement each background task. Each connection to the database results in the operating system spawning a process for that session. Client processes, such as sqlplus, connect to the TNS listener (tnslsnr), which does a `fork()` and then `exec()` in the database kernel program in `$ORACLE_HOME/bin/oracle`. The newly created process is known as a server process or shadow process.

On a Windows machine, there is a corresponding process (.exe) for each Oracle Database instance or SID. Each background task is a thread inside a single process. All background, server, and client processes are threads of a master Oracle Database process and all threads share resources.

For Sybase Adaptive Server Everywhere (ASE), connections to the database equate to tasks that are each assigned an engine. Prior to version 15.7, Sybase ASE executed each engine in a separate process. Starting with ASE 15.7, Sybase has offered a threaded kernel with each engine being a thread of a process and the engines communicating via shared memory. Engines run only user tasks, not I/O. Sybase has always used the thread-based model for ASE running on Windows platforms.

"The instinct for wanting to build a better mousetrap is endemic in the computing and software community"

Continual Evolution of Database Kernels

SQL platforms have gained a large share of the database market over the past three decades. The instinct for wanting to build a better mousetrap is endemic in the computing and software community. It's no surprise that we've seen the development of new architectures and new kernels for SQL, NoSQL, NewSQL and in-memory database platforms.

The original code base for MySQL (http://www.mysql.com/) dates back more than 15 years. Several factors (modern CPU architectures,

the growth of cloud computing, and a wide following for Web applications) contributed to a desire to refactor MySQL as a lightweight SQL DBMS.

Drizzle (http://www.drizzle.org/) is a leaner version of MySQL built using a microkernel architecture and plugins. To build Drizzle, developers have moved code from the MySQL kernel into plugins. MySQL found success with pluggable storage engines, such as InnoDB (http://www.innodb.com/). Drizzle extends the use of plugins for storage engines, authentication, replication, a function engine, string functions, catalog functions, schema dictionary, IPV6 functions, math functions, a multithread scheduler, and other features. When it starts up, Drizzle loads about four dozen default plugins.

Another interesting development is Monet, a main memory database kernel. Monet has roots in Troll, a relational kernel developed about three decades ago. MonetDB version 4 became an open source project in 2004. MonetDB (http://www.monetdb.org/Home) is an implementation of a column-store database kernel, a departure for traditional SQL databases that are row oriented. Although column stores are traditionally associated with analytics and business intelligence, MonetDB is capable of supporting ACID transactions. MonetDB has an algebraic database kernel that makes it versatile enough to support multiple query languages, including SQL and XQuery. It's based on a layered architecture that provides multiple interfaces for plugging in extensions. Support for math functions, strings, and temporal data types is provided by linked-in libraries. MonetDB is the subject of ongoing research and a recent interesting development has been DataCell (http://is.gd/ZysFSG), a project that provides sensor stream-processing capabilities built over MonetDB.

Another innovation has come from McObject, maker of the eXtremeDB embedded database product. McObject offers eXtremeDB-KM (http://is.gd/K4Nuey), a version that deploys in kernel mode to operate in the same address space as the operating system. This provides direct access of data structures to kernel processes, eliminating the overhead of context switches for applications requiring extreme performance.

Database Plugins

The extensible database manager, with an architecture that supports plugins, gives developers the ability to adapt a database to an applications' requirements. We can write pluggable indexing and storage engines, for example. We can also write plugins that extend the SQL language, but that is a subject for a future article.

Aside from Drizzle, we've also seen several NewSQL platforms (http://newsql.sourceforge.net/) that represent a refactoring of the relational database. Plugins and refactoring provide a reminder that the database is not an example of static technology.

— Ken North is a well-known expert in database technologies. He regularly contributes to Dr. Dobb's on the subjects of databases and software, including trends and techniques.

Comment

From the Vault

Dataflow Programming: Handling Huge Data Loads Without Adding Complexity

This popular article posted last fall on Dr.Dobbs.com shows you how to manage data pipelines rather than webs of threads and processes.

— *DDJ*

By Jim Falgout

Today, we speak casually of terabytes of data the way we used to speak of megabytes. Multi-petabyte data stores are common and growing. To handle all this data quickly and efficiently, and to extract useful business insights from it, we need programming paradigms that combine the scalable use of multicore processors and many-node clusters with data processing and analysis. No single programming paradigm can serve all problem domains, so expect many alternatives and consolidation. We're entering an era of innovation and the time is ripe for a robust programming paradigm well suited to big data processing and analytics. In this article, I'll discuss dataflow, a technique that manages very large data flows into and through islands of high-speed computation.

What Is Dataflow?

As a software architecture, dataflow consists of nodes in a graph connected by directed dataflow queues. The graph represents an application or program and the nodes represent functions to be applied to data. The dataflow queues transport the data between connected functions. Figure 1 shows connectors on the left reading data from data sources. The data flows from left to right, from node to node. In most cases, each operator in the graph receives data from its input queue, applies its function to the data, and outputs the results to its output queue. For example, the second operator, entitled "Missing Value," reads rows of data, looks for missing values in each row, and outputs replacement values to its output queue.

Composing a dataflow graph is a process of creating instances of operators and linking them together by stitching the output of one operator to the input of another. This composition model provides great flexibility in how users link operators to create applications. By thinking of each operator as a function consuming data and producing an output, dataflow provides the higher-order mechanism (composition) for organizing those functions.

"The dataflow model provides pipeline parallelism by its very nature"

Dataflow's shared-nothing model yields a simple programming paradigm, somewhat analogous to the UNIX shell where shell commands can be piped from one to the next to produce the desired output. Each shell command is independent but all live by a contract to consume standard input and pipe out standard output. Dataflow operators live by a similar kind of contract, using queues. Additionally, operators in dataflow can have multiple input and output queues. They also have properties that allow manipulating their behavior. Flow control is applied to queues to enable different consumption rates between operators. Deadlock detection can also be applied to prevent race conditions between operators.

The dataflow model provides pipeline parallelism by its very nature. As an operator consumes input it pushes its output downstream. Operators normally don't have to see all of their input before producing output. As each operator works, a pipeline of data makes its way through the graph. Each operator can be represented by a thread, or possibly a set of threads. With many operators in a dataflow graph working in parallel, dataflow readily takes advantage of multicore processors.

Dataflow also supports horizontal partitioning, allowing data to be segmented and applied to a replicated section of a graph. This partitioning can be dynamic, adjusting to the number of computing resources available at execution time. Horizontal partitioning is a great way to "divide and conquer" data problems where data dependency is not an issue. It can both scale up to multicore processors and scale out to multiple nodes within a cluster.

Dataflow History

One definition of dataflow frequently found in computer science textbooks describes an architecture whereby changing the value of a variable results in the automatic recalculation of other dependent variables. This is more of the "spreadsheet" model made popular by Microsoft Excel and others.

In his seminal paper titled "The Semantics of a Simple Language for Parallel Processing" (http://is.gd/y2LNmm [PDF]), Gilles Kahn defines Kahn Process Networks (KPNs). According to Wikipedia, KPNs define a programming model "where a group of deterministic sequential processes are communicating through unbounded FIFO channels" (http://is.gd/WI2RcW). This matches closely with how we've been describing dataflow.

KPNs were originally designed with distributed programming in mind. They are also heavily used for modeling signal-processing systems. The functional approach of KPNs is a good fit for signal processing, allowing filters to be built as pluggable functions that can be inserted as needed within a signal flow.

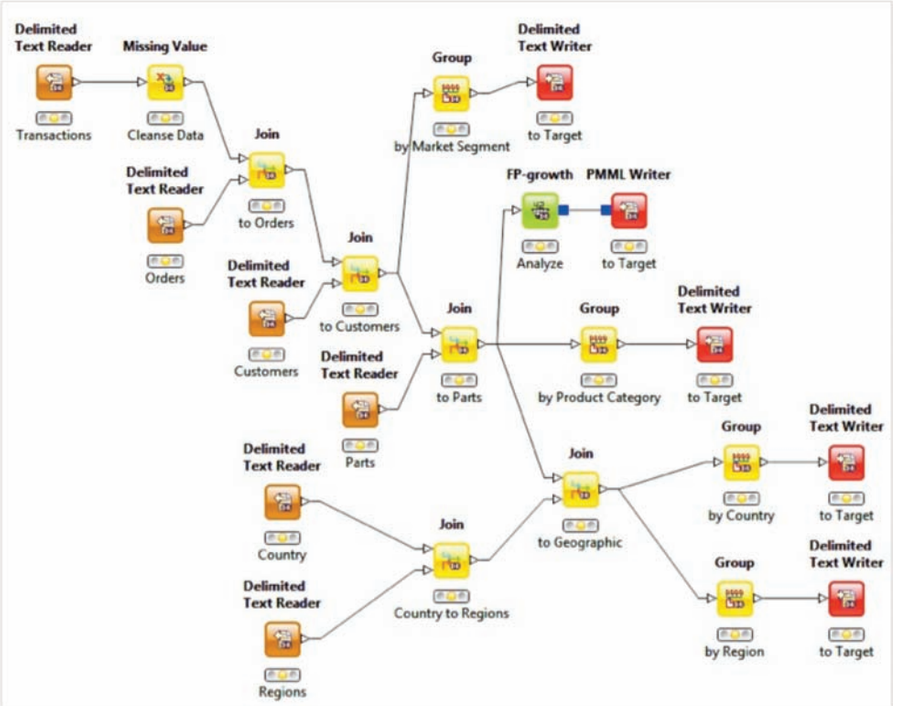


Figure 1: A dataflow graph processing and aggregating transactional data.

The original concept of dataflow (spreadsheets) and KPNs have, over time, evolved to the software architecture of dataflow.

Dataflow Goodness

The concepts of dataflow are easy to grasp leading to "design-time scalability." As Figure 1 shows, dataflow lends itself well to GUI environments, bringing high-performance computing to an extremely accessible usability level.

The java.util.concurrent library has been in existence since Java 5, yet few programmers have deep experience with it. With the proliferation of multicore processors and the abundance of raw data, developers need the ability to write high-performing, scalable software, which is almost impossible if we stay in today's thread-centric, shared-memory model of programming. Dataflow provides a means of composing scalable applications using a building block approach. It does so while abstracting away low-level issues such as threads, memory synchronization, and race conditions.

Dataflow has many of the good qualities of functional programming. For example, dataflow queues are immutable, eliminating concerns about shared memory synchronization or side effects from upstream operators. Immutability also allows queues to have multiple readers for good functional reuse. Having no global state also limits side effects. By contract, each operator in a dataflow graph only mutates the data it receives according to the function it provides. It's easy to look at a dataflow graph and surmise how it works and what it does. The similarity of this approach to the actor model cannot be overlooked.

Support for Big Data

Because the dataflow operators in a graph work in parallel, the model allows overlapping I/O operations with computation. This is a "whole application" approach to parallelization as opposed to many thread-oriented performance frameworks that focus on hot sections of code such as `for` loops. This addresses a key problem in processing "big data" for today's many-core processors: feeding data fast enough to the processors.

While dataflow does this, it scales down easily as well. This distinguishes it from technologies, such as Hadoop and to a lesser extent, Map Reduce, which don't scale downward well due to their innate complexity.

We've discussed how a dataflow architecture exploits multicore. These same principles can be applied to multi-node clusters by extending dataflow queues over networks with a dataflow graph executed on multiple systems in parallel. The compositional model of building dataflow graphs allows for replication of pieces of the graph across multiple nodes. Scaling out extends the reach of dataflow to solve large data problems.

"Actors define communication endpoints that allow other actors to find them and send them messages"

Dataflow and Actors

The actor model has been popularized by languages such as Erlang and Scala. In these models, independent actors communicate using messages. When an actor receives a message, it acts upon it as defined by functions inside the actor. Actors define communication endpoints that allow other actors to find them and send them messages. In some implementations, messages are free form and actors use regular expression forms to distinguish messages and how they are handled.

Below is a very simple example of an actor in Scala. This code is a fragment as it does not include imports or definitions of the message classes. The `MessageHandler` class extends the `Actor` library class, overriding the `act()` method. The `act()` method is invoked to handle incoming messages. The example code receives messages in a loop until a `Stop` message is received. Once a `Stop` is received, the sample invokes `exit()`, terminating the instance of the actor. Inside of the loop the receive block is used to process messages by type. The `case` statement is used per message type that can be received. Regular expressions can also be used to determine which messages are handled by a `case` statement. The `Message` type is handled by doing some processing of the message and then sending an acknowledgement to the message sender. The sender variable is defined by `Actor`.

```
class MessageHandler extends Actor {
    def act() {
        while (true) {
            receive {
                case Message =>
                    // Do some work to handle the message
                    // Send an acknowledgement
                    sender ! Ack
                case Stop =>
                    // Terminates the actor
                    exit()
            }
        }
    }
}
```

On the surface, the dataflow and actor models are very similar. They both support a functional, shared-nothing programming style. The concepts within each are easy to grasp. It is straightforward to pick up frameworks of either model and become proficient quickly. Both models are also easy to extend by writing new operators or actors. As with dataflow, the actor model hides multithreading issues from the developer. Having no shared memory means no synchronization worries.

As with dataflow, the actor model scales up as well as out. Both models also support an active or reactive implementation model. Active models imply an active thread per operator or actor. The reactor model

allows for a work-stealing implementation, where the number of system resources utilized can be contained, thus allowing for many thousands of instances of actors.

Below is an example written in more of a dataflow style. It is a simple operator that takes strings as inputs and outputs the length of each string as output. The looping construct is similar to an `Actor`, but handles a more static data type.

```
public class StringLength extends DataflowProcess {

    private final StringInput input;
    private final IntOutput output;

    public StringLength(Dataflow source) {
        // Handle properties, set up inputs and outputs …
    }

    @Override
    protected void execute() {
        while (input.stepNext()) {
            output.push(input.asString().length());
        }

        output.pushEndOfData();
    }
}
```

There are differences in the two models. The actor model does not guarantee immutability or message ordering; whereas dataflow does. Nor do actors optimize for large numbers of messages or messages with large amounts of data.

Dataflow provides flow control where the actor model doesn't (at least, not out of the box). Without flow control, messages either get dropped or memory usage problems occur as communication endpoints become full. Dataflow also generally does static binding of components at composition time, whereas actors are much more dynamic. Static binding enables graph analysis to apply optimizations.

The actor model appears much better suited to task-based parallelism where concurrency is important. For example, Erlang programs control telecom equipment with thousands of unique requests per minute. Meanwhile, dataflow is focused on data parallelism where single-job data processing and analysis performance are the main concerns. In a sense, they're slightly different solutions for different domains, although conceptually similar.

Performance Benchmarks

I used the MalStone B10 benchmark, which consists of 10 billion rows of log data in five fields, to measure performance of a dataflow-based framework. The overall data size is nearly a terabyte. The code for the benchmark extracts the year and week of the year from a timestamp field and aggregates a compromise indicator by site identifier, year, and week. A count of overall site activity is included and the summations are cumulative.

The benchmark was run on a single-node machine with 32 cores. Figure 2 shows runtimes for the benchmark. The test was run with increasing numbers of cores showing good scalability as the compute resources increased.

The test was also run on a 20-node cluster using Amazon EC2 instances. Each instance contained eight cores but less capable I/O systems. The overall runtime was 10 minutes, showing the ability to scale large processing jobs across a cluster using dataflow techniques.

Conclusion

Multicore and many-core will be the performance answer of processor providers for years to come. Software has yet to catch up, but mainstream evolution towards parallel, scalable software is taking place. As a software architecture, dataflow fits well in the big data domain. It enables impressive multicore and multi-node utilization and scalability. It's easy to learn and easy to use and scales both up and out well, as well as down for smaller workloads.

Further Reading

Simple Concurrency with Dataflow Variables in C++ (http://drdobbs.com/cpp/224202533)

— Jim Falgout is the chief technologist for Pervasive DataRush; Pervasive is a company that specializes in embedded, cloud-based, and highly parallel database technologies.

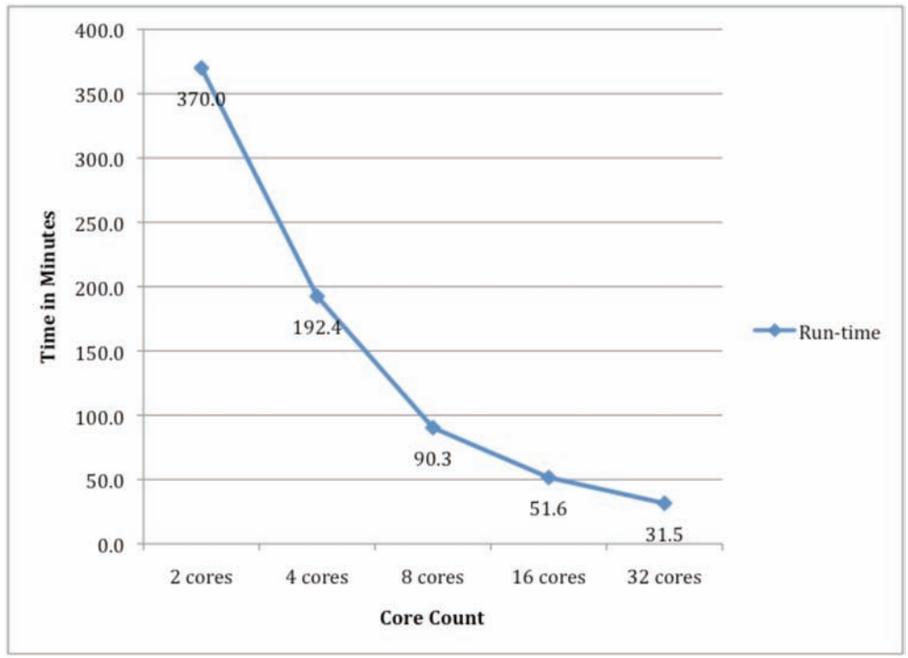
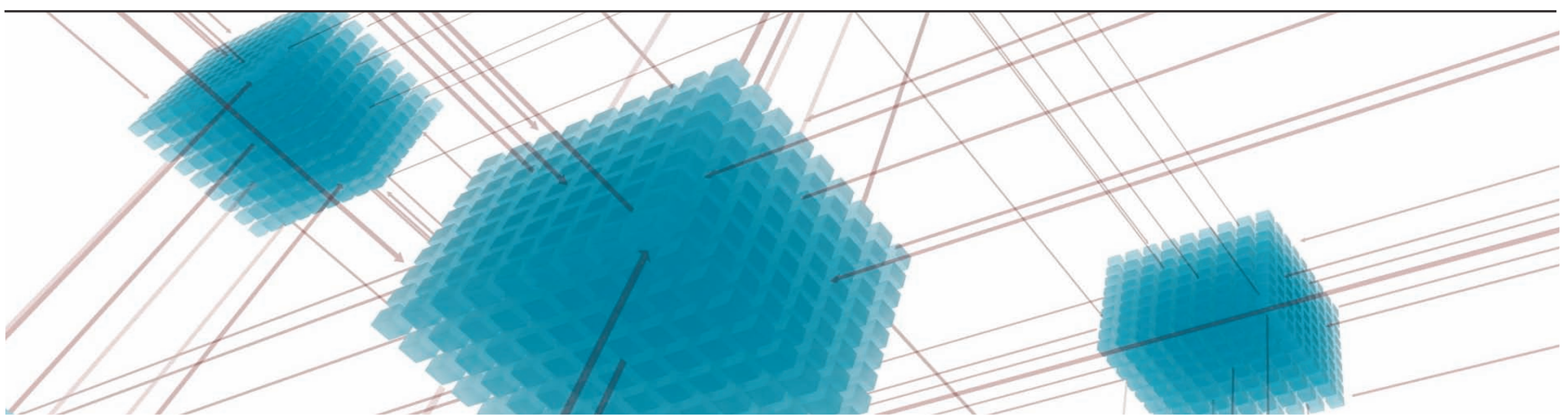


Figure 2: Malstone B10 runtimes using a dataflow framework.

Comment



This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

SIMPLE SEARCHING IN JAVA

Straightforward Java code to loop through and filter a list of entries as they are typed.

http://drdobbs.com/blogs/jvm/232700121

WRITING CHROME EXTENSIONS

Extensions to Chrome enable you to extend the browser's capabilities, change how pages are rendered, and interact with the server in a controlled manner. For all their power, Chrome extensions turn out to be comparatively simple to write and build once you know how.

http://drdobbs.com/mobile/232602420

IF SMALL TASKS ARE THE NEW PROGRAM UNIT FOR A MULTICORE WORLD, WHEN WILL WE ASSEMBLE PROGRAMS FROM THEM?

The right way to write multithreading code keeps evolving. If only we could keep up!

http://drdobbs.com/parallel/232602463

DEVELOPER'S READING LIST

C++ concurrency, web crawlers, Google testing, and more: This month's reading list is packed with great books on interesting topics.

http://drdobbs.com/tools/232900463

UNIFORM FUNCTION CALL SYNTAX

Walter Bright has become more and more convinced over the years that the key to code reusability and scalability is encapsulation — having subsystems communicate through small, well-defined interfaces and letting them hide their own implementation details.

http://drdobbs.com/blogs/cpp/232700394

CUSTOM CONFIGURING JAVA APPS: EXTENDING LOG4J

Creating an extended, server-side version of a log4j `Logger` that automatically configures itself and adds `printf`-like logging methods to the standard `Logger`.

http://drdobbs.com/web-development/232700307

IN THIS ISSUE

Dr. Dobb's

Andrew Binstock Editor in Chief, Dr. Dobb's
alb@drdobbs.com

Deirdre Blake Managing Editor, Dr. Dobb's
dblake@techweb.com

Amy Stephens Copyeditor, Dr. Dobb's
astephens@techweb.com

Sean Coady Webmaster, Dr. Dobb's
scoady@techweb.com

J. D. Hildebrand Editor at Large, Dr. Dobb's
jdhildebrand@drdobbs.com

Jon Erickson Editor in Chief Emeritus , Dr. Dobb's

DR DOBB'S
UBM TECHWEB
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

Dr.Dobb's Business Contacts

INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK

EVP of Group Sales, InformationWeek Business Technology Network, Martha Schwartz
(212) 600-3015, mschwartz@techweb.com

Sales Assistant, Salvatore Silletti
(212) 600-3327, ssilletti@techweb.com

SALES CONTACTS—WEST

Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

Strategic Accounts

Account Director, Sandra Kupiec
(415) 947-6922, skupiec@techweb.com

Account Manager, Vesna Beso
(415) 947-6104, vbeso@techweb.com

Account Executive, Matthew Cohen-Meyer
(415) 947-6214, mmeyer@techweb.com

MARKETING

VP, Marketing, Winnie Ng-Schuchman
(631) 406-6507, wng@techweb.com

Marketing Director, Angela Lee-Moll
(516) 562-5803, aleemoll@techweb.com

Marketing Manager, Monique Kakegawa
(949) 223-3609, mluttrell@techweb.com

Director, Client Marketing, Michelle Somers
(516) 562-7928, msomers@techweb.com

SALES CONTACTS—EAST

Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

District Manager, Steven Sorhaindo
(212) 600-3092, ssorhaindo@techweb.com

Strategic Accounts

District Manager, Mary Hyland
(516) 562-5120, mhyland@techweb.com

Account Manager, Tara Bradeen
(212) 600-3387, tbradeen@techweb.com

Account Manager, Jennifer Gambino
(516) 562-5651, jgambino@techweb.com

Account Manager, Elyse Cowen
(212) 600-3051, ecowen@techweb.com

Sales Assistant, Kathleen Jurina
(212) 600-3170, kjurina@techweb.com

AUDIENCE DEVELOPMENT

Director, Karen McAleer
(516) 562-7833, kmcaleer@techweb.com

BUSINESS OFFICE

General Manager, Marian Dujmovits

United Business Media LLC
600 Community Drive
Manhasset, N.Y. 11030 (516) 562-5000