

Embedded databases:

Why not to use the relational data model

By Duncan Bates



Handling data management the traditional way does not always translate well to embedded systems. The popularity of the relational database model is indisputable but does not mean it's the right choice when dealing with precious CPU, memory, and storage resources. One alternative to the relational model can help lower hardware requirements and model complex data relationships, allowing vendors to free up resources for the application at hand.

In an increasingly competitive market, embedded application vendors are constantly looking for new ways to cut application costs and time to market as well as increase application functionality to ultimately gain market share and boost product sales. While revenue margins are being squeezed, consumers are expecting higher quality and functionality from new product releases. Vendors are becoming more open to add third-party components to new and existing products to meet these goals.

One essential component in any embedded application is efficient data management. Commercial embedded data management engines are gaining acceptance and in many cases becoming a hard requirement for applications. As millions of dollars have been poured into R&D during the past 25 years, the relational model has emerged as a preferred method of data management.

Forming relationships

The overarching benefit of the relational data model is not the model itself, but rather its close relationship to the SQL language. SQL has two major benefits:

- **Ad hoc querying:** With a predefined relational data model, any valid SQL will guarantee a result without guaranteeing its performance. In data mining applications, this is an extremely powerful feature, but in most embedded applications, use cases and queries are known at design time. Think of an MP3 player: All use cases, such as music file synchronization and user navigation, are predefined and won't change until the next release of the device or firmware. It's not as if a manager would come and ask a developer to create a new report based on the existing data model.
- **Vendor independence:** SQL is a common language supported by many embedded database vendors. Presumably, replacing one with another should be as easy as flipping a light switch. Though it's not that simple, making this transition is definitely easier than moving from one proprietary API to another.

The relational model makes relationships between records by value matching and, in most cases, through keys. These keys are known as *primary key/foreign key relationships*. Figure 1 illustrates a relational model of artists and albums in an MP3 player, which will act as the basis for further discussion in this article.

Upon closer inspection, the relationship is implemented through copying the value of the fname into the album table and adding an index structure in between the two. Duplicating the fname field in itself adds overhead to the database image. Another implication in Figure 1 relates to the foreign key. If a foreign key data structure is not added, developers will have to visit each row in the album table every time they work their way through the relationship. The reason for this is that the table data is not in any order, so there is no way to tell if the matching value is at the beginning,

middle, and/or end of the table. Adding a foreign key index solves this table scan problem. Figure 2 breaks out the foreign key index and the album table to illustrate the index overhead. Most embedded database engines implement indexes as B-trees or any of the variants.

With the B-tree in place, developers can do a binary search to make the relationships, going from a table scan to an index scan. This translates moving from a linear search to a binary search, improving the cost of running through the relationship by an exponential difference.

A table scan costs $O(n)$ where n represents the number of records in the table, while an index scan costs $O(\log n)$. In computational complexity theory, big O notation is often used to describe how the size of the input data affects an algorithm's usage of computational resources.

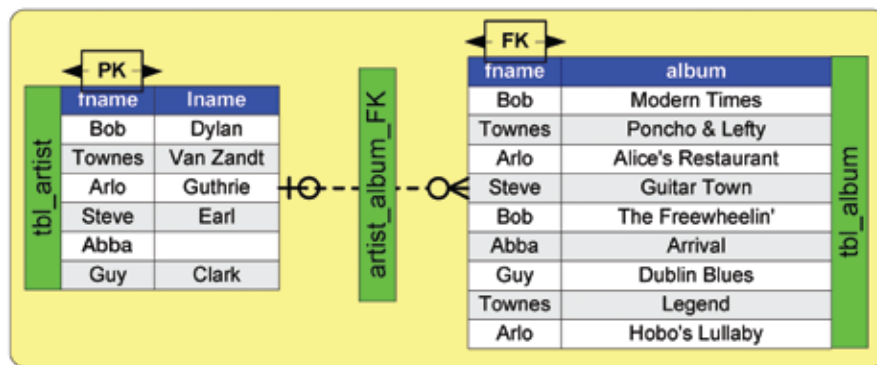


Figure 1

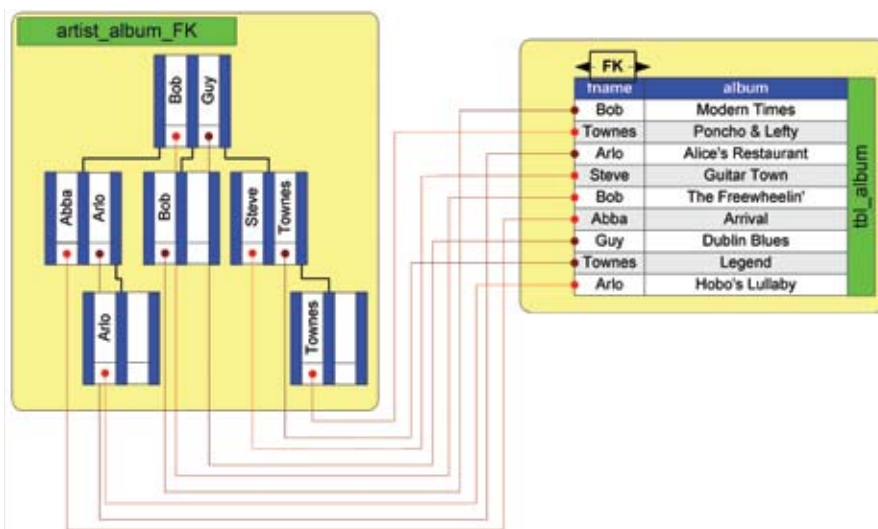


Figure 2



Other obvious implications include the space needed to represent the index and the hit taken to maintain this structure when data changes. Since I/O is the most costly operation from a time, CPU, and power consumption aspect, developers should strive to reduce it. With storage devices like flash, the writing should also be limited to deter a negative effect on the maximum write-erase cycles and space-reclaiming cycles imposed by the technology.

The question then becomes: How can developers maintain this relationship information with a cost less than $O(\log n)$?

Reintroducing the network data model

Figure 3 shows the same data representation adjusted through the network data model.

The network data model pre-dates the relational model and can be viewed as its superset. This implies that anything expressed in the relational model can be expressed in the network model, even SQL support. The main advantage is the way relationships can be modeled. In Figure 3, the relationships previously shown as a foreign key index are now broken down into multiple lists of pointers, called *sets*. Pointers can be viewed as void pointers in a C application with direct look-ups into the heap, except that the heap is now persistent storage. Eliminating the foreign key data structure and the fname duplication not only reduces the amount of data needed to store, but also unnecessary data structure maintenance.

The figure is simplified; an owner has two pointers, the first and last member record, while the member has three, the owner

plus the previous and next member. By a pointer's nature, it's not tied to any particular data type, so the relationships can model complex relationships between any number of record types, not only between two as imposed by the relational model. The complex modeling feature will not be discussed in this article, but it speaks to the network model's flexibility.

Cost implications

Moving from one record through the sets translates into constant cost. At most, a single I/O cycle is needed as long as the data is not already memory resident in the database RAM cache. With the foreign key implementation, the B-tree would have been traversed first, $O(\log n)$ cost, prior to locating the actual record. It's obvious that traversing the B-tree has CPU and I/O overhead, but it also has memory overhead. Any database cache

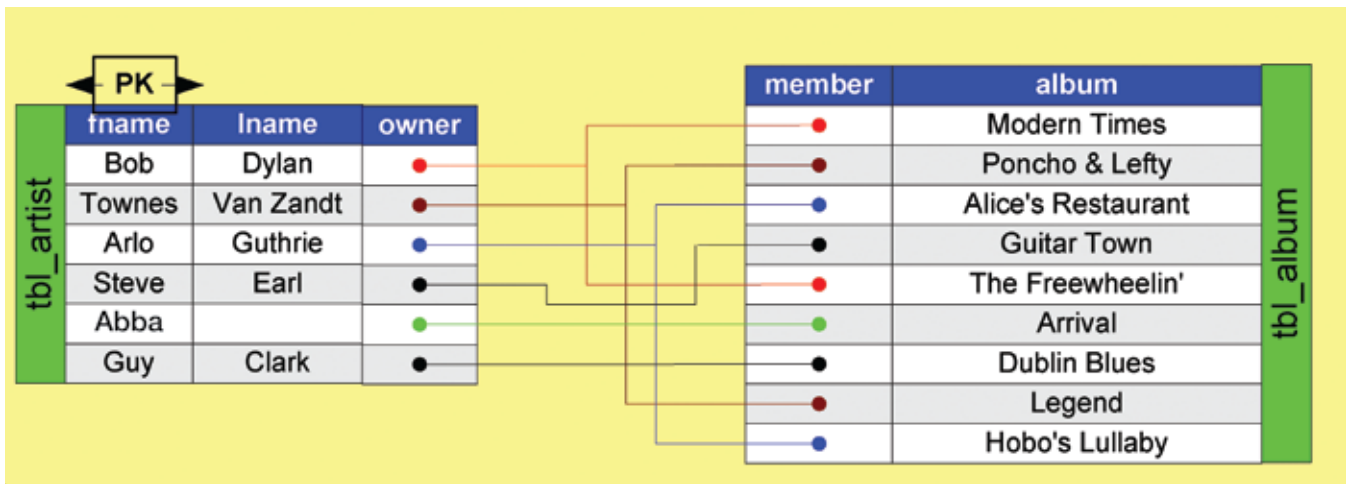


Figure 3

	Hardware			
	x86 desktop (34,000 records)		ARM7 consumer device (1,776 records)	
	Relational	Network	Relational	Network
Records inserted	81.62 sec	29.07 sec	193.88 sec	33.6 sec
Records updated	103.28 sec	15.28 sec	<i>Not Available</i>	<i>Not Available</i>
Records deleted	88.16 sec	17.30 sec	<i>Not Available</i>	<i>Not Available</i>
Records selected	15.81 sec	0.28 sec	1.25 sec	0.0012 sec

Table 1

will store the recently visited data, even B-tree data. Since the B-tree scan ends up in the cache, the cache must be large or suffer additional I/O to refresh its data.

Write operations also entail constant cost. A developer adding a new record to the album table and joining the new record to an existing artist-album set would need to take the following steps to accomplish the operation:

1. Add the new album record.
2. Set the new record's owner pointer to the current artist.
3. Set the new record's previous pointer to the current artist's last record.
4. Set the new record's next pointer to 0.
5. Set the current artist's last record's next pointer to the new record.
6. Set the owner's last pointer to the new record.

No scans are made during this sequence of operations, resulting in constant cost. With a B-tree implementation, a developer would:

1. Add the new album record.
2. Scan the B-tree to find the new record's index position.
3. If there is no room in the B-tree, split and reorganize the tree.
4. Write a reference to the new record in the B-tree.

During this sequence, the developer encounters an $O(\log n)$ cost in Step 2. More importantly, Step 3 potentially imposes a huge cost by requiring that parts of or the entire tree be reorganized. The reorganization is unpredictable because it depends on the fullness of the tree and where in the tree the change must be made. The more nodes that contain data, the

higher the chance of a larger reorganization. In most cases B-tree changes are done locally, affecting only a few nodes, but sometimes numerous nodes are touched, adding uncertainty to the application. Thus, if developers find themselves needing predictable performance, they should examine how their data is being represented.

An MP3 player benchmark

So how many hardware resources can a developer save using the network model? In one example, Birdstep Technology implemented a three-way relationship, artist->album->song, allowing a commercial MP3 player manufacturer to get some hard facts on resource savings. Developers carefully compared Birdstep Technology's RDM Embedded database engine, which is a network model, and a public domain relational database engine using both desktop computers and consumer electronics hardware. As Table 1 indicates, the more resource constrained the hardware was, the bigger the difference in savings.

On both of these hardware solutions, the network model used 27 percent less disk space to store the same amount of records and relationships. All the storage savings can be attributed to replacing the artist->album and the album->song foreign key indexes with pointers. Removing these data structures had a huge effect on the storage requirements. A B-tree index typically requires 1.3x the space of what it's indexing.

Application drives database decisions

When looking to add or replace existing data management components in an application, developers should consider the choices carefully. The application should drive the decision, not the industry. Several different solutions are available, ranging from simple libraries to full client server solutions, adding capabilities as described in this article. Selecting the right technology and modeling the data correctly can have a huge impact on the application's cost, resulting in more profitable revenue margins, higher-quality products, and a better end-user experience. **ECD**



Duncan Bates is VP of product management at Seattle-based Birdstep Technology, Inc., where he has worked since June 2001. Prior to joining Birdstep, Duncan maintained technical positions at LCC International, Inc., Ericsson AB, and other companies. He holds a Candidatus magisterii degree from the University of Oslo, Faculty of Computer Science.

Birdstep Technology
206-748-5245
duncan.bates@birdstep.com
www.birdstep.com/database