An Application-Specific Database

Kathleen Fisher, Colin Goodall*, Karin Högstedt, and Anne Rogers

AT&T Labs
Shannon Laboratory
180 Park Avenue
Florham Park, NJ 07932, USA
{kfisher, karin, amr}@research.att.com, cgoodall@ems.att.com

Abstract. *Signatures* are evolving profiles of entities extracted from streams of transactional data. For a stream of credit card transactions, for example, an entity might be a credit card number and a signature the average purchase amount. Signatures provide a high-level view of data in a transactional data warehouse and help data analysts focus their attention on interesting subsets of the data in such warehouses. Traditional databases are not designed for such applications. They impose overhead for services not necessary in such applications, such as indexing, declarative querying, and transaction support. Hancock is a C-based domain-specific programming language with an embedded domain-specific database designed for computing signatures. In this paper, we describe Hancock's database mechanism, evaluate its performance, and compare an application written in Hancock with an equivalent application written in Daytona [5], a very efficient relational database system.

1 Introduction

Recent advances in processing speed and disk technology have made it possible to process and store vast amounts of data, enabling new kinds of information applications. One such application involves mining daily transaction streams for information about the entities described in the transactions. Examples of such streams include AT&T's daily call-detail data, which contains one record per call, credit card transaction data, which contains one record per purchase, and TCPDUMP data, which contains one record per IP packet passing through a tap point. The entities of interest in these streams might be telephone numbers, credit card numbers, and IP addresses, respectively. If such transactional data simply flows into a data warehouse, it can be difficult to find which entities are "interesting" for a particular application because of the sheer volume of data. Where should data analysts choose to focus their attention?

One solution to this problem is to tap the transactional stream as it flows into the data warehouse and use the resulting information to build and maintain "signatures," which are small profiles of the entities in the stream [3]. Signatures evolve over time in response to additional transactions. Analysts design these

* Author's address: 200 S. Laurel Ave, Middletown, NJ 07748.

signatures to capture the "essence" of the entities mentioned in the stream along desired dimensions. These signatures serve as a high-level summary of the contents of a transaction warehouse and allow analysts to focus their attention on entities with interesting signatures. For example, a signature in the call-detail case might associate with each telephone number the average daily number of international calls made from that telephone number. Analysts might use this information to detect fraud, *e.g.*, if the number is higher than usual. AT&T has used signatures for a number of years to significant financial advantage [2, 3].

Local anecdotal evidence suggested that traditional databases were not suited to the task of building and maintaining signatures because they could not perform adequately. In particular, traditional databases had difficulty loading the daily transactions in a timely fashion[1].

Consequently, data analysts at AT&T wrote their initial signature programs using an *ad hoc* representation for their signature data that they custom designed for each program. They indexed their representation only by the identifiers associated with the entities they were tracking. To make their applications fault-tolerant, they took daily snapshots of their signature collections, providing a coarse-grained roll-back mechanism. Their implementations were well-tailored for the desired class of applications, and they met performance requirements set by the analysts and the consumers of their data.

Despite the success of the data produced by the initial signature programs, the analysts were not satisfied with the programs themselves because they were hard to write and maintain. Although the per-entity code was conceptually very simple (count the number of international calls, *etc.*), the code to manage the volume of data efficiently was not. This infrastructure code swamped the per-entity code, making programs difficult to write and maintain. Analysts had written a handful of successful applications using this technique, but the complexity of the programs dissuaded them from writing more. Maintenance was a problem because the analysts had to review the programs periodically to ensure that they complied with current federal regulations. Hence although analysts thought signatures very useful, they were at a loss as to how to compute them: traditional databases were too slow, and the *ad hoc* approaches were too complex.

In response to this problem, we designed Hancock, which is a C-based domain-specific programming language with support for a domain-specific database embedded within it. The goal of the Hancock project is to make it easy to read, write, and maintain programs that compute signatures from transactional data. All of the original signature programs have been re-written in Hancock, and the new versions have been running in production for several years now. In addition, analysts have designed new signatures using Hancock, and the maintenance problem is now much simpler.

An earlier paper [2] describes the Hancock language; in this paper, we focus on Hancock's domain-specific database mechanism and its performance. In Section 2, we discuss the performance requirements for signature applications. Section 3 presents Hancock's `map` abstraction, which is the language interface to Hancock's database mechanism. We describe the implementation of maps in

Section 4 and performance results in Section 5. In Section 6, we compare the performance of a signature application written in Hancock with the performance of the same application written in Daytona[5], a relational database system tuned for high-volume applications. Finally, we conclude in Section 7.

2 Requirements

Maintainable signature programs are useless if their performance fails to meet the requirements of their user base. Programs written in Hancock must satisfy requirements regarding the time necessary to process a new batch of transactions, the amount of disk space necessary to store the signatures for a given application, and the time necessary to lookup the signatures associated with various collections of customers. We discuss each of these requirements in turn.

We must be able to process all the transactions in a batch before the next batch arrives so that we do not fall behind. In addition, we must have sufficient head-room to allow us to catch up if machine maintenance, data transmission errors, or disk failures cause system down-time. Signature collections must be space efficient, because given extra disk space signature researchers can *always* devise new information to compute.

The third performance constraint imposed on the Hancock system involves the time required to query data stored in signature collections. There are four different ways in which users access this data, each with its own time constraint. The first type of access arises when a person types an entity identifier into a web interface. This person expects to receive the associated signature within web time — a second or so. The second type of usage involves retrieving the data for collections of several hundred thousand identifiers. Analysts feed such lists into the system and expect to have the associated signatures within the length of time necessary to get a cup of coffee — roughly five minutes. The third access pattern occurs when an analyst wishes to examine the signatures of all the entities stored in a signature collection, perhaps to look for anomalies or to compute aggregates. For such computations, the analyst needs to be able to make several such passes over the data in one day; hence these computations can take no more than an hour (and preferably less). We call the fourth type of access to signature data *unordered* references. Unlike typical accesses, a sequence of unordered references does not exhibit good spatial locality. Such references occur during signature construction when data analysts look up the signatures of secondary identifiers appearing in the transactional data. Although unordered references are less important to our application class than the previous three access patterns, they should be supported to the degree possible without violating any of the other performance constraints.

3 Hancock maps

Hancock provides an abstraction, called `maps`, for representing persistent collections of signatures. Unlike traditional databases, Hancock maps provide an

extremely limited set of operations: associating a signature with a key, retrieving or updating the signature associated with a key, removing a key from the database, asking if a given key has an associated signature, and iterating over a range of keys with stored signatures. To avoid the associated overhead, Hancock maps do not support transactions, locking, secondary indices, or declarative querying. In the remainder of this section, we describe the functionality of maps in more detail.

3.1 Map declarations

A Hancock programmer describes the structure of a map using a map type declaration, which has the following form:

```
map sig_m {
  key id_t;
  value sig_t;
  default SIG_DEFAULT;
};
```

This declaration specifies a new map type named `sig_m`. We describe each of the clauses in this declaration in turn.

The `key` clause indicates that `sig_m` maps will be indexed by values of type `id_t`. Such values typically represent some form of identification number, for example, telephone numbers, credit card numbers, or IP addresses. Programmers determine, in part, the underlying structure of a map through the map key specification. We will discuss the connection between map keys and map structure in Section 4.1 after we explain how maps are implemented.

The `value` clause specifies the type of data to be associated with each key. This type can be any C type of statically-known size. We refer to a key with a value in a given map as an *active* key and to an active key and its associated value as an *item*.

The `default` clause specifies a value to be returned when a programmer requests data for an inactive key. The default may be a constant, as is `SIG_DEFAULT` in this example, or a function. A constant default must be a constant expression that has the value type of the map. A function default specifies the name of a function that computes a default value for a given key.

Hancock maps provide generic compression for their values. In some cases, application programmers may have domain-specific knowledge that enables them to write custom compression functions that are significantly better than the defaults. So, in addition to the `key`, `value`, and `default` clauses, Hancock allows the programmer to specify optional `compress` and `decompress` clauses, which specify functions to compress and decompress values. Hancock allows compression functions to use variable-width schemes, because they often yield better compression ratios.

Variables can be declared using the usual C syntax (for example `sig_m m`). To connect a map on disk with a Hancock variable `m`, we must associate the name of the file containing the map with the variable. Hancock provides *initializing declarations* to make this connection. For example, the following statement,

`sig_m m = "09-Aug-2001.m";`, declares that the variable `m` has type `sig_m` and can be found on disk in the file named `"09-Aug-2001.m"`.

3.2 Map operations

Hancock provides four operations for manipulating items in maps: read, write, test, and remove. Hancock's indexing operator, written `<: ... :>`, can be used as an r-value (for reading) or as an l-value (for writing). Hancock's test operator, written `@<:..:>`, queries a map to determine whether the key is active. Finally, Hancock's map remove operator, written `\<:..:>`, removes an item. As an example, the following code first reads the value for the key `id` from map `m`, writes a different value `newsig` into map `m` for the key `id`, tests whether the key `id1` is active, and if so removes that item.

```
sig_m m;
id_t id,id1;
sig_t oldsig, newsig;

oldsig = m<:id:>;      /* Read id's data */
   ⋮
newsig = ...
m<:id:> = newsig;     /* Write id's data */

if (m@<:id1:>)        /* Test id1  */
  m\<:id1:>          /* Remove id1's data */
```

Hancock also provides a mechanism for performing an operation on every active key in a map by combining Hancock's `iterate` statement, which allows a programmer to consume a stream of data, with an expression for creating a stream from a map. The expression `m[startKeyExpr..stopKeyExpr]` generates a stream of active keys from `m` that fall between the value of the expressions `startKeyExpr` and `stopKeyExpr` inclusive. A detailed discussion of Hancock's iterate statement can be found in the Hancock manual [4].

4 Implementation

Hancock's map implementation is a variant of a data structure used in AT&T's Global Fraud Management System. The original data structure, which mapped phone numbers to values, was designed to support efficient updates and efficient queries for single keys. This data structure split each key (a phone number) into two pieces: a primary key, which identified a *block* of data, and a secondary key, which identified a particular *entry* in a block. The primary key was used to index into an array that gave the location of the corresponding block in the file. The values in the blocks were of uniform size and the blocks were fully populated by using a (constant) default value for inactive keys. The design allowed the location of an entry to be computed directly from the location of the block.

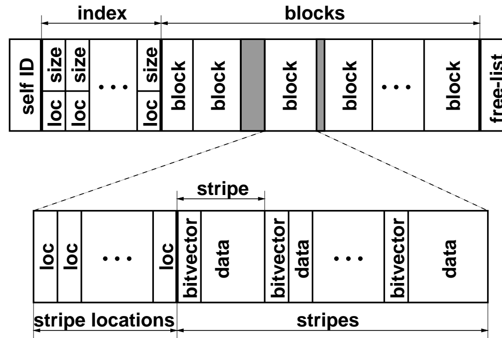

Fig. 1. The on-disk data representation of a Hancock map.

This data structure worked well for references that had good locality. Analysts obtained the desired locality for transaction streams by sorting the records by the key. Given ordered records, each block in the file would need to be read/written at most once during an update. The design also worked well for single-key queries because each such reference would take exactly two disk reads: one to consult the index and one to read the block. Despite its strengths, this data structure was not sufficient for our purposes because fully populating the blocks wasted space and did not support unordered references efficiently.

Our design, which is essentially a multi-level table,[1] uses compression to reduce the disk space needed to store maps. The programmer chooses the unit of compression based on the characteristics of an application. It breaks the blocks into sub-blocks, which we call *stripes*, and includes a stripe index in the block. To read an entry from a block, the runtime system uses the block index to find the block in the file, reads the stripe index from the file, uses the stripe index to find the stripe, reads the stripe, decompresses it, and finally, returns the desired value.

Our design also uses caching to address the problem of unordered references. One class of applications that generates unordered references uses a second key from a transaction as a map index during an update. These references tend to show some locality, especially when a significant portion of the working set can be retained. To address the problem of unordered references, our implementation caches recently-accessed compressed stripes.

Figure 1 shows the on-disk representation used by our implementation. It includes a self-identification field, a block index, an unordered set of blocks, and a free-list for unused space. The self-identification information is used to match the file against the declared type of a map to help prevent users from inadvertently using a map with the wrong type for an application. Each block contains a stripe index followed by the stripes, in order. The in-memory representation, which is not shown, includes only the index and a cache of recently accessed stripes.

---

[1] Other applications, such as paging or IP address lookup for routing[6–8] use variants of a multi-level table to support large key spaces.

4.1 Defining key types

In Section 2, we mentioned that Hancock programmers determine, in part, the structure of a map. In particular, programmers determine the size of the primary index and the stripe size for a map by specifying the type of the key, which must have the following restricted form: the key type must be a C `struct` type, it must have three fields, and those fields must have Hancock `range` types. For a value with such a type, the first field corresponds to the block number, the second to the stripe number within that block, and the third to the particular element in that stripe. The fields are required to have `range` types (integers in a given range) to bound the number and size of the blocks and stripes.

The main tradeoff to consider when deciding the block size is the size of the index (12 bytes per entry on disk, 20 in memory). Choosing a stripe size, on the other hand, requires weighing various considerations, including the cost of decompressing values, the size of a decompressed stripe in relation to the processor's primary and secondary cache sizes, and the expected mix of access patterns for the data.

5 Performance results

In this section, we describe experimental results that show that the performance of Hancock maps meets the criteria presented in Section 2. We start by describing our experimental set-up.

The experiments were performed on one processor of an SGI Origin 2000. The processor, an R12000, has split 32KB instruction and data caches and an 8MB secondary cache. The machine has more than 6GB of main memory.

We report *real* time that was computed using the Unix `time` utility. `Time` reports the elapsed time the measured command spent running (`real`), the CPU time spent in execution of the command and its children (`user`), and the CPU time spent executing system code by the command and its children (`system`).

We use two Hancock maps in our experiments. The first, called *activity*, maps phone numbers to three-byte values. The sample map we use has more than 464M active keys. It uses a simple compression function that compresses three-byte values into two bytes. The second, called *features*, maps phone numbers to 124 byte values. The sample features map we use has 163M active keys. The user-specified compression function for this map generates a variable number of bytes per record.

Both maps use the key type show in Figure 2. This type uses the first six digits of a phone number, called the *exchange*, as the primary key, while the last four digits (called the *line*) are split between the secondary and tertiary keys depending on the stripe size (`MAXENTRY`).

In Section 2, we described a set of performance requirements for signature programs. We must be able to process all the updates for a given day in one day with enough headroom to allow for system-related problems (such as network downtime). We must use disk space efficiently. And we must be able to

```
#define MAXEXCHANGE 1000000
#define MAXLINE 10000
#define MAXENTRY 100
#define MAXSTRIPE (MAXLINE/MAXENTRY)

 range npanxx_r = [0 .. (MAXEXCHANGE-1)];
 range stripe_r = [0 .. (MAXSTRIPE-1)];
 range entry_r = [0 .. (MAXENTRY-1)];

struct key_t {
  npanxx_r primary;
  stripe_r secondary;
  entry_r tertiary;
};
```

Fig. 2. Key type for activity and features maps.

answer single-key queries in one second, work-list queries in five minutes, and map iteration queries in under an hour.

In this section, we discuss Hancock's performance in light of these requirements using a stripe size of 100 entries for the two maps; we selected this size because it has the best overall performance.

To determine the cost of accessing a map during a daily update, we measured the performance of a Hancock program that performs one map read and one map write for each call in an ordered daily call stream. We chose this restricted program because different applications have widely different per call computation costs that are unrelated to the performance of Hancock maps. Using a call stream with more than 275M calls, the restricted programs took roughly 19 minutes for the activity map and 30 minutes for the features map. For comparison purposes, a program that simply consumes the stream took slightly less than eight minutes.

Hancock maps are reasonably space-efficient. Representing each item directly would take eight bytes for the activity map (five for the key and three for the value) and 129 bytes (5+124) for the features map. The activity map, which is 1.27GB in size, uses 2.7 bytes per item including the overhead from the block index, the stripe location vectors, and the bitvectors. The features map is 1.10GB in size. It uses a much more aggressive compression function (it also has more data to work with) that consumes 6.7 bytes per item including overhead.

Table 1 presents a summary of map query performance. We ran each experiment multiple times, but we report the first or *cold* times, because they were higher than all the other runs and they best approximate the experience of a user. The single-key experiment (`single`) requested the data for a single key. The work list experiments requested data for an unordered list of 156,051 keys that we obtained from a colleague (`unordered`). The performance data for the ordered case (`ordered`) include the cost of sorting the keys (11s). The map iteration experiment (`iteration`) iterated over the map reading the data for each active key in the map, but not performing computation on the extracted values.

Table 1. Summary of cold query times for a stripe size of 100 entries.

Query type	Activity	Features
`single`	1.1s	1.1s
`unordered`	209.9s	242.7s
`ordered`	11s + 25.3s	11s + 37.4s
`iteration`	1023.1s	738.4s

In summary, Hancock maps have reasonable update performance, reasonable disk usage, and meet the query performance requirements from Section 2.

6 A database comparison

This section compares the performance of two different implementations of a simplified production application for analyzing calling-card records. We wrote one implementation in Hancock and another in Cymbal, the programming language provided by Daytona[5], a highly efficient database management system.

Although the obvious point of comparison for Hancock would have been an SQL implementation for some popular commercial database system such as Oracle, we did not do such an experiment because we wanted to be as fair as possible. To that end, we selected Daytona rather than a commercial database because Daytona is highly efficient both for loading large volumes of information on a daily basis[2] and for supporting fast queries on that data. Its efficiency makes it the preferred database at AT&T for production data analysis applications. We used Cymbal rather than SQL, which Daytona also supports, because Cymbal programs can be highly tuned and a quick, back-of-the-envelope experiment convinced us that an SQL implementation would be much too slow to be competitive with the Hancock implementation.

The application we use for our experiments maintains usage statistics for a subset of the calling cards seen on the AT&T network. Each day, the application uses the records that describe the day's calls to compute statistics for each card, including the number of calls, the number of zero-length calls, the number of call attempts (short calls), the total duration of all calls, and the total (approximate) charge. This application uses a ten-digit canonicalized version of the calling card number as the key. It associates seven day's worth of usage statistics with each such key. The call records, which include the card number, the date of the call, the call duration, and the standard rate (charge) for the call, are stored in a Daytona database.

[2] A Daytona database currently holds more than a year's worth of call detail data. The largest table in this database contains over 100 billion rows and will continue to grow until it reaches about 200 billion records in August 2002.

6.1 The Hancock implementation

The Hancock implementation cannot access the call records directly because they are stored in a Daytona database. Instead, we wrote a Daytona program to pull the calls from the database and produce a stream of C structs, one per call. The Hancock implementation reads this stream from a UNIX pipe. An alternative implementation would pull the calls from the "raw" feed before they were entered into the database. In practice, we use this approach for many Hancock applications, but for this experiment we used data that was already stored in Daytona. For most of our experiments, we report times that include the cost of pulling the calls from the database,, but we give some results in Section 6.3 that show the performance of Hancock without the cost of pulling the calls from the database.

```
typedef struct {
  int calls;
  int zeros;
  int attempts;
  int secs;
  int rate;
} usage_t;

typedef struct { usage_t u[7]; } vusage_t;

int statsCompress(vusage_t *block, unsigned char *to, int to_ext);
int statsUncompress(unsigned char *from, int from_size, vusage_t *to);

 map cnStats_m {
  key cnbr_t;
  value vusage_t;
  default { {0,0,0,0,0}, ... };
  compress statsCompress;
  decompress statsUncompress;
}
```

Fig. 3. Definition for `cnStats` map.

The usage statistics are stored in a Hancock map with the definition shown in Figure 3. The ten-digit keys are decomposed into physical keys of type `cnbr_t` as follows: the first five digits are used as the primary key or block number, the next two as the stripe number, and the remaining three as the entry number.

The implementation of the usage collection program is straightforward: sort the input stream by the canonicalized card number, which puts all calls charged to a given card adjacent in the stream. As each new card is seen, look up the current value for that card in the map. Use each call record to update the usage

statistics as appropriate. After seeing the last record for a card number, put the the final result back into the map.

6.2 The Daytona implementation

Daytona is a commercial-grade database system. It provides fully-indexed and horizontally-partitioned data tables with several levels of locking, and it supports parallel processing. Daytona's programming language, Cymbal, supports both procedural and declarative programming, including an extension of SQL. Programs written in Cymbal are translated to C and then compiled and linked to runtime libraries.

After some trial-and-error, we implemented a strategy for computing usage that is both efficient and reasonably expressible in Daytona. We store the usage statistics in a Daytona database containing one record for each card number. Each such record contains one field for the card number and 35 integer-valued HEKA-coded[3] fields for the computed statistics. The records are represented on disk in ASCII with the fields separated by pipes and the records separated by newlines. On average, these records take 51 bytes.

The computation has three steps:

1. Loop over the call records, growing and updating an associative array indexed by card numbers.
2. Loop over the entire usage database, updating in-place the entries for keys that have records in the associative array (*i.e.*, cards that had calls in the stream). Mark records in the associative array as they are incorporated into the database. Update the index.
3. Loop over the associative array, appending a new entry to the end of the database for each unmarked record. Update the index.

Looping over the database to do the updates guarantees that access to the database will be sequential.

We report results for two versions of the Daytona implementation: one sequential and one parallel. The parallel program is very similar to the sequential program. The usage database includes ten horizontal partitions. Each process in the parallel program is assigned one of ten partitions of the key space. Each process reads each call record, but only processes those records with keys in its partition. Cymbal contains constructs that make it easy to parallelize Daytona programs in this way.

6.3 Experimental results

In this section, we report on the performance of the Hancock and Daytona versions of our application. We performed our experiments on a production SUN Ultra Enterprise 10000 Starfire with 64 250-MHZ processors and 30GB of memory. As a vehicle for timing experiments, this machine is less than ideal because

[3] HEKA is a compact encoding supported as a basic type in Cymbal.

varying contention for resources yields non-uniform timings. However, we had to use it as our experimental platform because it is where the call database is kept.

The first set of experiments computes the usage database for seven days of calls, one day at a time. Most days had two million calls, except days one and two, which had 1.1 million calls. Table 2 contains running times for updating the usage information. We ran each experiment three times (except those marked with an asterisk, which we ran twice). We report the minimum `real` time for the three runs. We also report the `user` and `system` time that corresponded to the minimum `real` time and their sum. Note that the sum of `user` and `system` time may be larger than the `real` (or elapsed) time because the application may run on multiple processors in parallel. We report the minimum because there was a high variance for the sequential Daytona implementation. The Hancock implementation showed at most a 10% difference in running time between runs, while timings for the sequential Daytona implementation differed by as much as 30%, in part because the machine's load varied more during these experiments.

Comparing the `real` times reported in Table 2, the Hancock program outperforms the sequential Daytona program, but it is slower than the parallel Daytona program for the daily computations. The sum of `user` plus `system` time accounts for this difference in resource consumption. Using this metric, the Hancock program outperforms both Daytona versions after the first day.

Table 2 also gives the maximum memory footprint corresponding to the run with the minimum `real` time for the three implementations. Hancock requires substantially less memory than the Daytona implementations; the difference becomes more pronounced later in the week. The memory usage of the Hancock programs is a function of the map key types used in the programs. As a result, it is constant across the days. The memory usage of the Daytona programs is bounded by the size of the associative array and the number of previously active card numbers that are updated during the second phase of the computation. Later in the week, more of the card numbers that appear in a given day will be in the database already and generate updates rather than appends, which explains why the memory footprint grows as the week progresses.

Table 2 also shows the disk space used to store the usage statistics database. The disk space for the index in the Daytona implementations varied slightly between runs. We report the disk space for the sequential run with the minimum `real` time. The Hancock maps are substantially larger than the Daytona databases (up to 30%) for the early days of the week, but the situation reverses towards the end of the week. Hancock's map implementation has more up-front overhead than Daytona's database implementation, but Hancock has a smaller per-item incremental cost. The table does not include the temporary disk space used by the Hancock implementation. It uses between 36 to 65MB for sorting. Plus the temporary space needed for "yesterday's" map.

Computing weekly usage database The daily experiments reported in the previous section reflect how these program would be used in practice, but the amount of data involved is quite small. In this section, we report performance

Table 2. Performance for computing the usage database for seven successive days. Results are reported for the minimum `real` time over three runs (except those marked with an asterisk which we ran twice).

Hancock version

	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6
`real`	8m49s	8m3s	8m2s	10m14s	10m0s	9m54s	9m57s
`user`	8m31s	7m28s	7m22s	9m27s	9m8s	9m9	9m8s
`system`	4m15s	3m11s	3m0s	5m8s	5m15s	4m58s	4m58s
`user+system`	12m46s	10m39s	10m22s	14m35s	14m23s	14m7s	14m6s
`memory`	123.8MB	123.8MB	123.8MB	123.8MB	123.8MB	123.8MB	123.8MB
`disk space`	64.4MB	80.3MB	91.4MB	106.8MB	119.2MB	129.8MB	139.3MB

Sequential Daytona version

	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5*	Day 6*
`real`	10m52s	15m5s	16m37s	25m52s	31m44s	37m1s	37m42
`user`	9m52s	9m35s	10m4s	13m37s	14m13s	15m28s	15m11s
`system`	0m48s	5m50s	6m23s	12m4s	17m18s	22m46s	23m43s
`user+system`	10m40s	15m25s	16m27s	25m41s	31m31s	38m14s	38m54s
`memory`	214.7MB	172.5MB	184.4MB	273.9MB	279.6MB	298.4MB	297.0MB
`disk space`	51.5MB	61.3MB	74.5MB	105.6MB	120.5MB	133.3MB	146.3MB

Parallel Daytona version*

	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6
`real`	7m6s	6m22s	6m15s	7m9s	6m42s	6m36s	6m36s
`user`	6m39s	53m28s	43m9s	50m12s	55m54s	53m58s	54m29s
`system`	0m11s	2m18s	4m21s	5m16s	8m32s	11m54s	13m20s
`user+system`	6m50s	55m46s	47m30s	55m38s	64m26s	65m52s	67m49s
`memory`	175.5MB	144.4MB	250.2MB	249.1MB	441.9MB	472.6MB	558.6MB

results computing the usage data using a full week's worth of calls in one run (see Table 3). Please note that due to a mistake when collecting the data for this experiment we used a slightly different version of the Hancock program that uses less memory, but has similar performance. While still modest, this experiment does show some issues that will arise as we scale to larger applications.

As in the previous experiment, the parallel Daytona version has the best performance, but it uses substantial resources. The Hancock program outperforms the sequential Daytona program on the `real`-time metric, but not on the "sum of `user` and `system`"-time metric. The Daytona program that pulls calls from the database is a bottleneck for this computation. We measured the effect of the Daytona portion of the computation by running only the Hancock program, using a file to hold the pre-pulled calls. The column labelled "Hancock Only" in Table 3 gives the result of this experiment, showing that the Hancock program

outperforms both Daytona implementations when drawing its data from disk. In our experience, this approach is how Hancock programs are used in practice: data is pulled once from a database, sorted, and then used by many Hancock applications (or Hancock receives a "raw" data feed directly).

Table 3. Performance for computing usage statistics from one week's calls, starting with an empty database. Time reported is the minimum `real` time over three runs (except those marked with an asterisk, which we ran twice). Reported memory size is the maximum memory size for the minimum `real`-time run.

	Hancock + Pull	Hancock Only	Sequential Daytona*	Parallel Daytona*
`real`	34m4s	20m35s	46m52s	23m42s
`user`	33m41s	8m6s	43m51s	4h4m50s
`system`	26m7s	24m41s	4m23s	5m3s
`user+system`	59m48s	32m47s	48m14s	4h9m53s
memory	72.0MB	65.5MB	587.1MB	1622.4MB

6.4 Summary

Large data management applications, such as computing signatures, require good locality of reference to perform efficiently. Our Hancock and Daytona implementations for computing signatures achieve good locality in very different ways, leading to different resource usage profiles. Our Daytona implementation uses associative arrays to store intermediate results and then makes updates in database order at the cost of substantial processor and memory resources. In contrast, our Hancock implementation sorts the data to obtain good locality of reference at the cost of the sort and temporary disk space. We should note that it would be possible, if somewhat unnatural, to implement the algorithm used in the Hancock program in Daytona and vice versa. By removing the difference in algorithms, we could learn more about the implications of using Hancock maps as contrasted to a conventional database table with indices.

We omitted our experiments on query performance for this application because of space constraints, but both Daytona and Hancock had no trouble meeting the guidelines.

For our, admittedly modest, experiment, the approach used in the parallel Daytona implementation is faster than the sorting approach used in the Hancock implementation, but it requires much more memory and more processors. The question of which system would work better for computing signatures on the full volume of AT&T's daily call detail data, which is two orders of magnitude larger than our calling-card application, remains. In practice, Hancock has shown itself to work well for such applications. We do not know whether

the Daytona approach will handle larger scale signature applications, but the amount of memory needed for our modest application suggests that it may not scale much further as is.

7 Conclusion

In this paper, we described a domain-specific database embedded within Hancock, a language for computing signatures. Hancock makes designing and maintaining programs such programs easier, while meeting a set of performance criteria for daily load, individual query, work-list query, and map iteration times.

The paper provides a modest comparison with Daytona, a high-performance database, by writing a simple application in both systems. The results of our experiment are somewhat mixed. Given unlimited computing resources and a sophisticated Cymbal programmer, Daytona is somewhat faster for the card-usage application. The Hancock implementation takes slightly longer and requires more temporary disk space; however, it requires substantially less memory, fewer processors, and less programmer expertise.

Hancock is available for non-commercial use from the Hancock website: `www.research.att.com/projects/hancock`.

References

1. David Belanger, Kenneth Church, and Andrew Hume. Virtual data warehousing, data publishing, and call detail. In *Processings of Databases in Telecommunications 1999, International Workshop.*, 1999. Also appears in Springer Verlag LNCS 1819 (pp. 106-117).
2. Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith. Hancock: A language for extracting signatures from data streams. In *Proceedings of the Sixth International Conference on Knowledge Discovery and Data Mining*, pages 9–17, August 2000.
3. Corinna Cortes and Daryl Pregibon. Giga mining. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, 1998.
4. Kathleen Fisher, Karin Högstedt, Anne Rogers, and Frederick Smith. Hancock 1.1 manual. See www.research.att.com/projects/hancock, 2001.
5. Rick Greer. Daytona and the fourth-generation language Cymbal. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*. ACM Press, 1999. Also available at www.research.att.com/projects/daytona.
6. P. Gupta, S. Lin, and M. McKeown. Routing lookups in hardware and memory access speeds. In *Proc. 17th Ann. Joint Conf. of the IEEE Computer and Communications Societies*, volume 3, pages 1240–7, 1998.
7. N.-F. Huang, S.-M. Zhao, J.-Y. Pan, and C.-A. Su. A fast IP routing lookup scheme for gigabit switching routers. In *Proc. 18th Ann. Joint Conf. of the IEEE Computer and Communications Societies*, volume 3, pages 1429–36, 1999.
8. B. Lampson, V. Srinivasan, and G. Varghese. IP lookups using multiway and multicolumn search. *IEEE/ACM Transactions on Networking*, 7(3):324–34, 1999.