♦ Sunrise: A Real-Time Event-Processing System

Gerald D. Baulier, Stephen M. Blott, Henry F. Korth, and Avi Silberschatz

Conceived at Bell Labs, Sunrise is a high-throughput, real-time event-processing and aggregation system. It offers many of the data-management features of a conventional database system, such as high-level, declarative programming interfaces and the traditional atomicity, consistency, isolation, and durability (ACID) correctness guarantees for transactions. Adapted to meet the needs of high-performance applications, Sunrise maintains its high throughput by using a shared-nothing parallel architecture that can deliver close to linear scale-up in practice. Sunrise achieves the real-time responsiveness (on the order of as little as a few milliseconds per transaction) that many emerging applications require by relying on the DataBlitz™ main-memory database system as its underlying single-site storage manager. The DataBlitz storage manager guarantees the fast access to correct data expected from a database system. The strength of Sunrise lies not only in its performance, but also in its flexibility. This paper introduces Sunrise and its two major components: a real-time analysis engine (RAE), at the heart of the runtime system, and a service authoring environment (SAE), which is the Sunrise authoring interface.

Introduction

The performance of transaction-processing systems has traditionally been measured in terms of the "number of transactions per second." An increasing number of applications, however, face real-time performance constraints. For the most part, the performance needs of real-time applications are met by custom-designed information systems. Such custom solutions work well, but they make it impossible to amortize the cost of a system over a large number of applications. As the number of applications has grown, the custom-solution approach has become economically untenable. This paper describes the Sunrise system, a flexible general solution to the real-time information-system problem.

One major source of applications demanding real-time information processing is telecommunication, which has always needed real-time guarantees for setting up telephone calls. Until recently, however, most calls could be set up by applying a straightforward mapping from a phone number to a physical telephone. A variety of new features and services are now making this mapping more complex. The increasing use of toll-free numbers (800 or 888 numbers in the U.S.) and call forwarding, along with the advent of local-number portability and the popularity of wireless systems with roaming capabilities, are causing a rapid increase in the number of calls that require number translation or redirection. Moreover, number-translation algorithms, which were previously restricted to table lookups, are becoming more sophisticated.

As these applications evolve, new features and services are being introduced that require processing to be performed at the time a call is set up. Among these are debit-based billing (requiring an on-line check of a customer's balance before completing a call), fraud detection and prevention, and route selection based on customers' profiles (such as quality-of-service guarantees, or the use of the Internet, voice

network, or a virtual private network). These new features and services are part of the intelligent network. Within the next few years virtually every call will be an intelligent network call.[1] This level of demand for high throughput, real-time response, and "intelligent" processing is beyond the capacity or capability of current systems.

A further limitation of current systems is their lack of flexibility. Until the late 1980s, the type of processing required to set up a phone call was well defined and relatively static. But escalating competition in the long-distance market has led to competitive pricing and discounting schemes (such as "Friends and Family"*), which are subject to frequent change. These complexities have continued to grow with the introduction of local competition and the merging of the Internet, various private networks, and the traditional voice network into a single communication system. In this environment, change is constant. Thus, flexibility requires more than modifiability; it requires the ability to alter some or all aspects of call processing without any system down-time to install those changes.

This paper focuses on the implications of these changes on information systems and on Sunrise, a system conceived at Bell Labs to meet the demands of this new environment. Sunrise can be used with any event-processing environment, including telecommunication, electronic commerce, and Internet service provisioning. Within telecommunication, Sunrise provides a basis for enhanced billing systems, fraud detection and prevention, local-number portability, settlements among service providers, and real-time traffic analysis, among others. The remainder of this paper describes the architecture of Sunrise and details its salient features. These include the real-time analysis engine (RAE), at the heart of the Sunrise runtime system, and the service authoring environment (SAE), which is the Sunrise authoring interface.

Panel 1. Abbreviations, Acronyms, and Terms

ACID—atomicity, consistency, isolation, and durability
CDR—call detail record
CPU—central processing unit
DBMS—database management system
GUI—graphical user interface
I/O—input/output
IP—Internet protocol
RAE—real-time analysis engine
SAE—service authoring environment
SAL—service authoring language
SQL—structured query language
TCP—transmission control protocol

An Overview of Sunrise

Many transaction-processing systems have performance requirements that cannot be met by conventional database systems. In telecommunications, for example, a variety of adjunct switching services such as debit-based billing, number mapping, call forwarding, and local number portability involve transaction processing during the critical call-setup phase of a telephone call. To meet the real-time requirements of the network, the service time for such events must not exceed a few milliseconds. In conventional database technology, however, the costs of invoking a structured query language (SQL) operation over a client-server interface, or the costs of a single access to secondary storage, can already account for hundreds of milliseconds. As a consequence, performance goals on the order of a few milliseconds may be unattainable even before the costs of a transaction's logic are taken into account.

Because of these limitations, many high-performance systems are based not on general-purpose database management systems (DBMSs), but rather on custom database systems. Custom systems are tightly coupled to their particular applications and tuned to the specific requirements of those applications. These solutions generally work well in practice and, from the perspective of performance alone, can be close to optimal. Along with these advantages, custom systems also have serious disadvantages. The cost—in monetary terms—of developing and maintaining them can be high and generally cannot be amortized across a number of applications. Moreover, custom systems are frequently inflexible, making it difficult, or even impossible, to adapt them to unforeseen or evolving requirements.

Sunrise, a general-purpose event-processing system, was developed to strike a balance between the

performance benefits of custom databases and the flexibility and maintainability of conventional database systems. On the one hand, the critical path for event processing within Sunrise has been specifically adapted to achieve high performance—even at the expense of functionality—if that functionality is of questionable value for real-time systems (see, for example, the discussion of data storage for processed events, later in this paper). On the other hand, Sunrise retains many features of conventional database systems, including high-level, declarative programming interfaces, and the correctness properties of atomicity, consistency, isolation, and durability (ACID) for transactions, as described in **Panel 2**. These features enhance the reliability, robustness, usability, and maintainability of both Sunrise and the applications built on Sunrise.

Figure 1 shows the typical configuration of a Sunrise-based system. Sunrise processes events on behalf of some real-time element (such as a network switch) and maintains summary and aggregation data over those events. To meet real-time performance goals, all the data necessary for event processing is stored in the DataBlitz™ main-memory database system. Because of space limitations in main memory, it is not possible to store information about individual processed events within Sunrise itself; instead, processed-event records are typically sent to a data warehouse for archiving. Archived data might then be used later for non-real-time tasks, such as auditing, data mining, and reprocessing (if all processing cannot be performed when the event occurs). Sunrise has been built to work with a commercial relational DBMS (for example, the Oracle* DBMS) as its back-end data warehouse.

This distinction between real-time data and archived data is fundamental to the Sunrise approach. Sunrise requires space complexity to be bounded over any sequence of events, regardless of the number of events in the sequence. This assumption limits the class of processing that can be supported. For example, a telecommunications pricing plan such as "The Most"* cannot be realized wholly within Sunrise, because it has unbounded space complexity. This plan awards a special discount for the phone number a customer calls most in a particular billing cycle.

Panel 2. ACID Transactions

The correctness properties of atomicity, consistency, isolation, and durability (ACID) for transactions are as follows:

- *Atomicity*. Either all or none of the effects of operations in a transaction are reflected in the database.
- *Consistency*. A single transaction executed against a database in a consistent state transforms that database into another state, which is also consistent.
- *Isolation*. Each transaction appears to be executed in isolation, even if other transactions are in fact executed concurrently.
- *Durability.* After a transaction is successfully completed, the changes it has made to the database persist, even if there are system failures.

All of these assume that each transaction transforms the database from a *consistent* state to a new *consistent* state.[2-4]

Determining which number that is, however, requires maintaining statistical information that grows with each new number called. In general, Sunrise stores configuration data, summary data, and aggregation data. *Configuration data* is read-only data that supports event processing. It might include, for instance, rating tables, customer information, or routing information. *Summary data* provides condensed information about processed events, such as histograms, or counts of events satisfying some property. *Aggregation data* combines information such as averages, minimums, and maximums over all processed events. In general, summary and aggregation data are updated as events are processed, but configuration data is not.

Figure 2 shows the two major components of Sunrise, RAE and SAE:

- RAE is a lean, single-site database-system kernel adapted to meet the needs of high-throughput, real-time systems. It achieves real-time responsiveness by using the DataBlitz main-memory database system and high throughput from its shared-nothing architecture, which can run several instances of RAE in parallel.

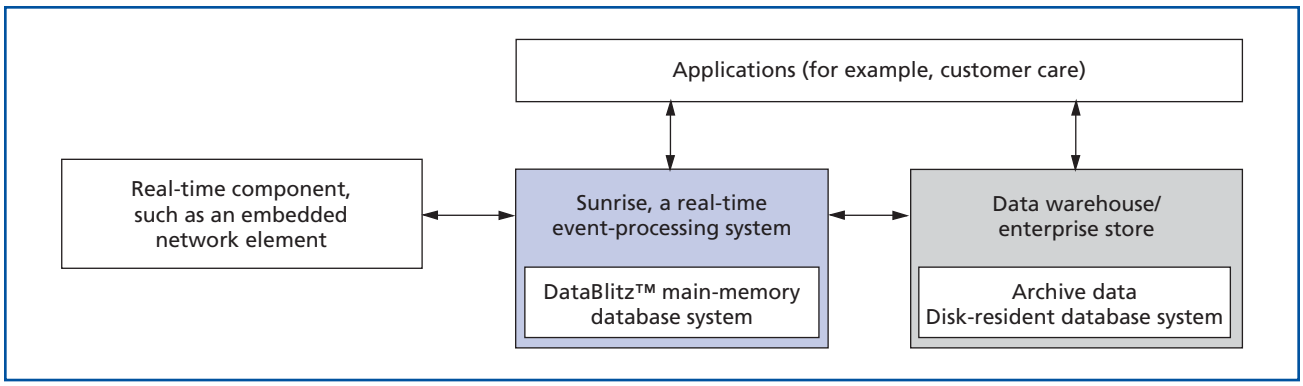


Figure 1.
A typical Sunrise configuration.

- SAE is a tool for authoring application-specific services and embedding those services within RAE. These services define:
  - The event-processing logic of a system;
  - The configuration, summary, and aggregation information that is maintained to support event processing;
  - Canned queries; and
  - The input and output streams that enable Sunrise to interface to existing data sources or sinks (frequently network elements or a data warehouse).

  Newly created services—or existing services, adapted to new requirements—are compiled and can be installed on line without interrupting event processing. In this way, Sunrise remains available, even as event-processing logic evolves over time.

RAE is the real-time event-processing and aggregation engine of Sunrise. Because RAE processes events on behalf of network elements with real-time performance requirements, it frequently receives its workload in the form of records transmitted over a set of streams, either from a single data source or several data sources. RAE processes an event using a set of application-specific services, which are authored through the SAE. The set of services invoked for event processing is subscription based, and the subscription model (described later in this paper) provides a compromise between the flexibility of ad hoc queries and the performance of canned queries. The side effects of event processing are updates to summary and aggregation information within the memory store and outputs (if any) transmitted over output streams. Outputs generally pipe data or results back to one of three locations:

- The network element that Sunrise serves,
- An application-specific client (for queries), or
- A data warehouse (for archiving processed-event records).

One of the strengths of Sunrise is its flexibility. The SAE provides a set of tools for authoring and maintaining application-specific services. Authoring is a high-level procedure, based on a set of graphical user interfaces (GUIs) and a fourth-generation service authoring language (SAL). As such, Sunrise is applicable to a wide range of areas, including billing systems, intelligent networks, Internet services, and network management. Possible specific applications in telecommunications include debit-based billing, fraud detection, call centers, real-time bill calculation, and adjunct switching services such as local-number portability and toll-free number mapping.

Example 1: Debit-based billing. *Consider a debit-based billing system for telephone calls, where each customer has a predeposited balance. A* `callSetup` *event occurs whenever a call is placed. The goal of processing* `callSetup` *events is to establish a preapproved duration for a call based on the customer's current balance. If sufficient funds are available, then a maximum preapproved duration is determined. If insufficient funds are available, then a preapproved duration of zero is assigned, and the call is effectively blocked. When the call is completed, a* `callCompletion` *event occurs, at which point the actual charges for the call are calculated and debited from the customer's balance.* □

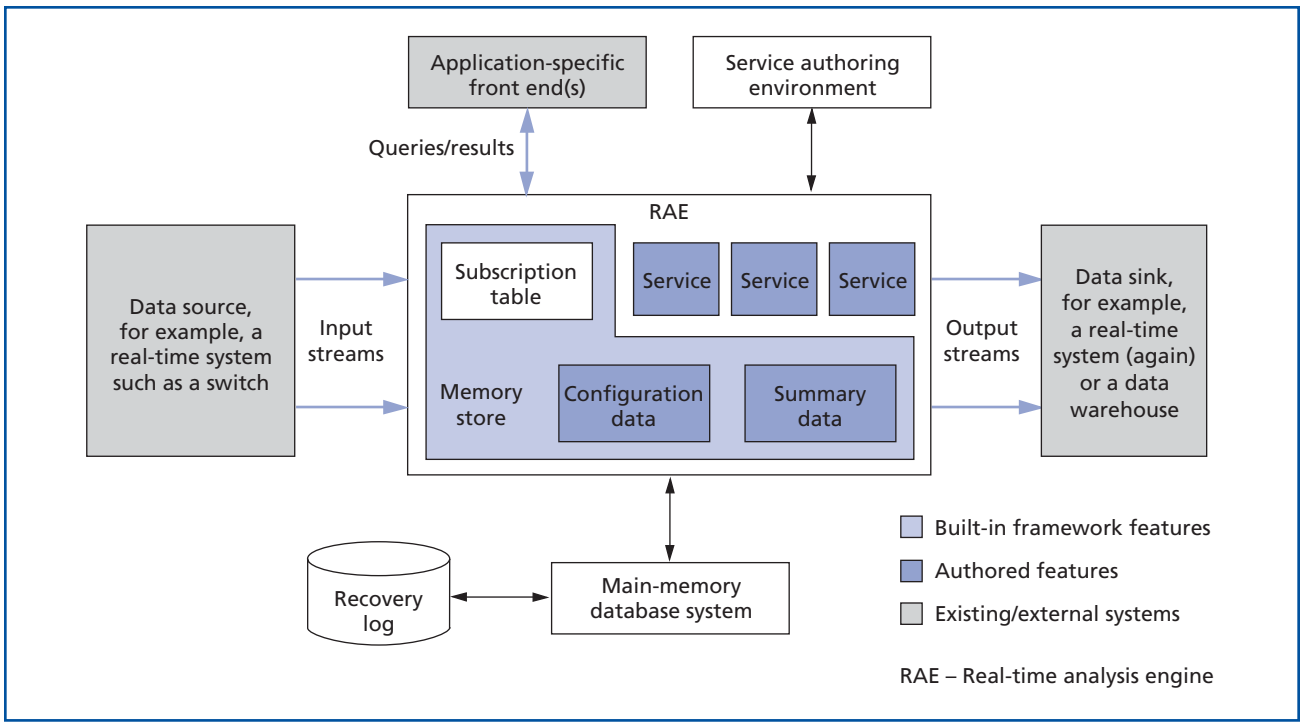


Figure 2.
Static architectural sketch.

The debit-based billing system of Example 1, which is closely coupled to the provisioning information and switching elements within a network, must be highly available. Because `callSetup` events are processed during the critical connection phase of a telephone call, they must meet the real-time performance requirements of the network. These typically dictate that the response time for event processing must be only a few milliseconds. The need for high throughput is also clear. Many network elements handle peak rates of many hundreds of calls placed every second.

However, performance is not the only issue raised by this example. From the perspective of functionality, computing the charges for a call is not trivial, generally depending on the set of discount plans subscribed to by a customer. In practice, the specific plans offered by carriers are subject to frequent change, and individual customers subscribe to not one, but several, plans. For example, different charges may apply to local, long-distance, and international calls. Moreover, the charges may depend on such things as the day, the time of day, the length of the call, and/or the volume or value of calls generated by the customer at hand. Almost all calls are also subject to taxation. Some plans, such as those based on a customer's volume or value, depend on aggregation data, which must be maintained as events are processed. Over time, both the algorithms for computing charges, and the configuration, summary, and aggregation data on which those algorithms are based—such as rate tables and customer summaries—evolve. As such, a debit-based billing system must offer both performance and flexibility.

Example 2: Fraud detection/prevention. *A fraud detection system may function as follows. The system maintains summary information, termed a* fraud signature, *describing each customer's regular calling pattern. For example, one customer may make frequent international calls, while another seldom places calls outside his or her state or local exchange. Similarly, one customer may generally place calls on weekday mornings, while another is more active on weekends. Fraud signatures describing these usage patterns are stored for each customer. Signatures are then used to detect irregular patterns—with similarity to known fraudulent patterns—when and if these occur.* □

This second example requires a processing model similar to the one used in Example 1, but the summary information and event-processing algorithms it

uses are very different. In particular, fraud signatures are highly dependent on the class of fraud being targeted. International fraud may require signatures describing patterns in the destinations of calls, whereas patterns in the sources of calls may be more relevant for mobile fraud. As such, the algorithms for maintaining and comparing fraud signatures can be complex. Moreover, fraud patterns themselves change over time, and fraud signatures and their associated algorithms must evolve to reflect those changes.

The Real-Time Analysis Engine

The role of RAE—the single-site, event-processing kernel within Sunrise—is similar to that of a conventional database server. To meet the performance goals of Sunrise, however, the architecture of RAE has been adapted in several respects.

The Memory Store

A key component within RAE is its *memory store*. The memory store is persistent and offers all the ACID guarantees of a conventional database system. It stores the configuration, summary, and aggregation data that supports event processing. As such, the performance of the memory store is critical to the performance of RAE as a whole.

To meet real-time performance requirements, the memory store is based on the DataBlitz main-memory storage manager, which offers transactional access to persistent data at main-memory speeds. (Descriptions of the DataBlitz storage manager and its architecture have been published previously under its prototype name, Dalí.[5,6]) In conventional, disk-resident database systems, only a small portion of a database is buffered in memory. The rest is accessed from secondary storage if and when it is required. A single disk access can account for from tens to hundreds of milliseconds, making real-time performance somewhat of a lottery. The DataBlitz storage manager, on the other hand, has been designed and finetuned under the assumption that the entire database resides in main memory. This assumption is becoming attractive for many applications as memory prices fall and machines with gigabytes of main memory become increasingly affordable. Moreover, the data structures, algorithms, and architecture of the DataBlitz storage manager are designed for main-memory residency, which improves performance still further.[7-9]

The DataBlitz main-memory storage manager also incorporates several special features for on-line transaction processing. A technique known as *ping-pong checkpointing* allows a consistent database state to be recorded on nonvolatile storage without interrupting regular transaction processing.[5,10] The DataBlitz storage manager supports hot spares, which provide availability, even in the presence of single-site failures. A hot spare is run parallel to and up to date with a primary site. Should the primary site fail, the spare is available to take over its workload (more or less) immediately, thereby shielding applications from many of the effects of single-site failures.[3]

Embedded Services

In client-server databases, part of a transaction's work is performed at the database client and part at the database server. Partitioning the task into two parts is satisfactory for many applications, but it introduces considerable overhead. Invoking a database operation is costly. Even if the client and server are on the same machine, there are still overhead costs associated with marshaling arguments and results across the server interface and with context switching between the two processes. Before the server executes ad hoc queries, it must also parse each query, generate execution plans, and select the best execution plan. With a secondary-memory database, these overheads are usually considered acceptable, since performance is normally dominated by input/output (I/O) costs. In Sunrise, however, these overheads cannot be neglected because database operations are processed at main-memory speeds.

Sunrise is not a client-server system in the traditional sense. Instead, application-specific services and RAE execute within the same process address space. Moreover, Sunrise does not admit ad hoc queries, so all the costs of selecting a query plan are incurred once, statically. This embedded approach reduces costs by eliminating communication costs and shortening path lengths. If a transaction first accesses data in the memory store, then performs some application-specific processing, and finally performs an update again within the memory store, it does so wholly within the context of a single process or thread.

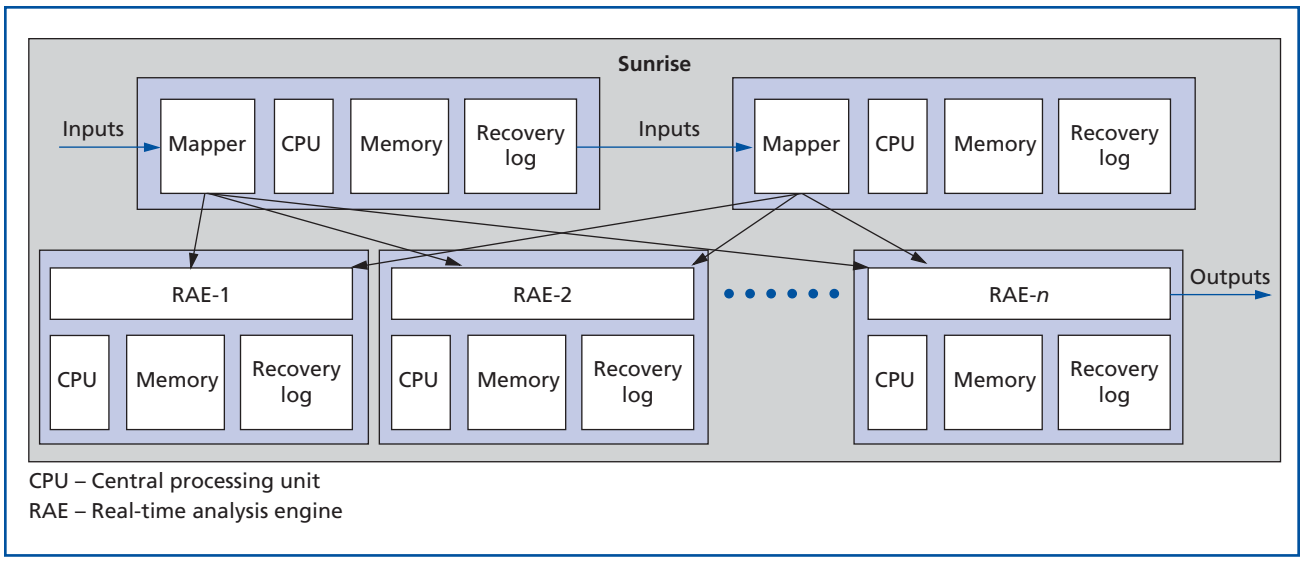


Figure 3.
Parallelism: several RAEs processing transactions in parallel.

Although embedding services in this way improves performance, it also introduces a number of potential pitfalls, one of which is safety. If service code were to raise any kind of unforeseeable error condition—such as a memory leak, a segmentation violation, or an infinite loop—it would compromise the integrity of RAE itself. In its mildest form, this might lead to resource leakage. More worrisome, however, is the scenario that RAE might crash and become unavailable for a period of time, or that the memory store might even become inconsistent (because of incorrect transactions or corrupted or lost data). Sunrise addresses this difficulty by using the SAE. The SAE provides high-level, declarative programming tools that can validate event handlers statically and compile services in a way that either avoids or handles the error conditions mentioned above. This mitigates many of the safety risks associated with embedding application-specific code within RAE itself.

Distribution and Parallelism

The two key resources governing the performance of RAE are main memory and central processing unit (CPU) cycles. If overstretched, both of these can become bottlenecks. The available memory limits the number of customers whose summary and aggregation data can be placed in the memory store. However, since workload is generally associated with customers, the number of customers that can be supported is also bounded by the available CPU cycles.

If a single instance of RAE does not have sufficient resources for a particular workload, Sunrise allows multiple RAEs to work in parallel, as shown in **Figure 3**. Each instance of RAE has its own CPU, its own memory, and its own instance of the memory store (with its own recovery log). Every memory-store table is either replicated across all sites or partitioned across sites. Generally, configuration data such as rate tables are replicated, whereas summary and aggregation data are partitioned. This shared-nothing approach to parallelism harnesses the aggregated memory and CPU resources of several machines. Shared-nothing parallelism is well suited to real-time, "rifle-shot" transaction processing, and throughput scale-up can be close to linear. As such, doubling the number of processors can double the throughput while still maintaining the same average response time.

The key components in this architecture are the *mappers*, which assign events to the instance of RAE on which they are processed, as shown in Figure 3. A mapper maintains a mapping table, which is implemented as a hash table. The mapper uses one of two rules to map an event to an individual RAE for processing. Under the first rule, an entry in the mapping table for the event's subscriber identifies the RAE to which the event is assigned. If there is no entry in the

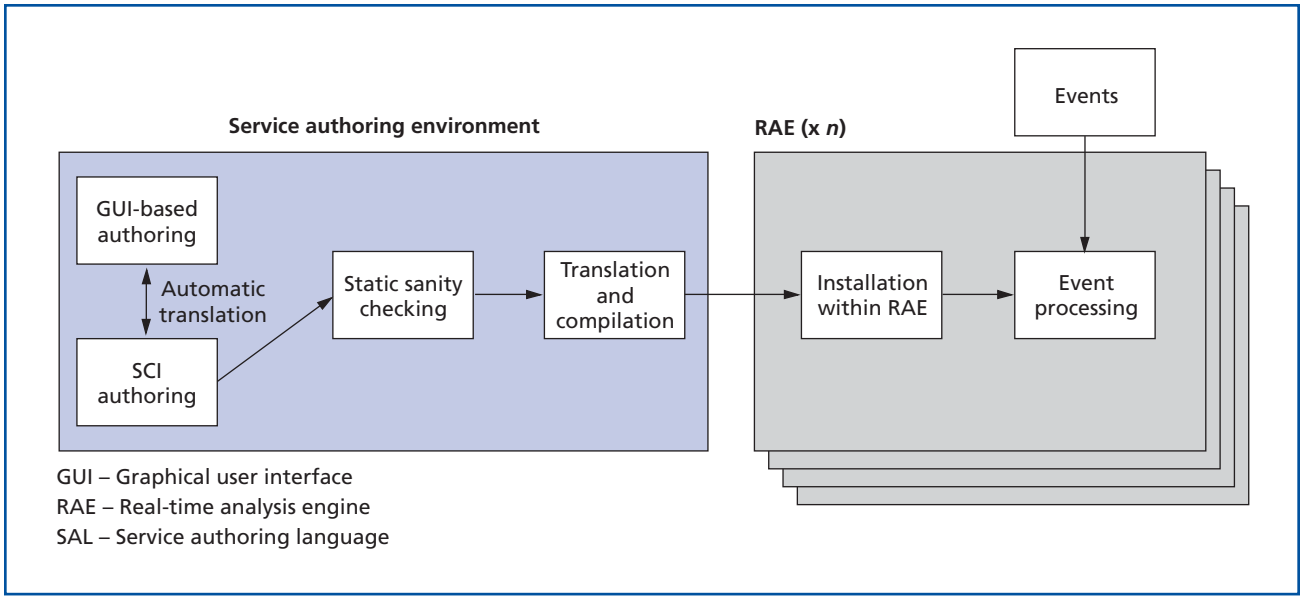


Figure 4.
SAE: service authoring, validation, compilation, and installation in RAE.

mapping table, then the second rule uses a hash function to select the RAE to which the event is assigned. An explicit mapping table allows subscribers to be relocated if there is a load imbalance between RAEs, or if one subscriber must be colocated with another (for example, to support customer hierarchies). However, the memory overhead of maintaining an explicit mapping table can be avoided for the (common) case in which the default—hash-function mapping—suffices. For events whose processing spans several RAEs, Sunrise manages distribution automatically, keeping it transparent to applications.

The Service Authoring Environment

Sunrise is not an application in its own right, but rather a platform on top of which event-processing applications are created. SAE, the tool that supports this authoring process, provides facilities for authoring:

- Input and output streams;
- Event-processing logic;
- Configuration, summary, and aggregation data necessary to support event processing; and
- Canned queries that can be invoked within a system.

Authoring these features is a high-level procedure, based on a set of GUIs and a fourth-generation, largely declarative language, SAL.

SAL inherits many of its data management features from SQL.[4] While, in principle, service authoring might have used a general-purpose language such as C++, Perl, or Java,* there are many practical advantages to using a richer, though more restrictive, declarative approach. With respect to performance, declarative languages are more amenable to automatic optimization, parallelization, and complexity analysis.[11,12] For example, if several RAEs are running in parallel, then the processing of a single event may span several sites. Such issues of distribution are transparent to authors, which in a sense is the essence of the declarative approach. Other advantages include simplicity and safety of authored services.

Figure 4 illustrates the authoring, compilation, and installation procedures for SAL scripts. Whether authors choose the GUIs provided or prefer to author services in SAL directly is purely a matter of taste. In either case, the result is a set of authored services, as described by SAL scripts. When authors submit these scripts, SAE performs a variety of sanity checks to ensure their correctness. For example, SAL is statically typed, allowing a broad class of programming errors to be detected at compile time. Static type checking also eliminates the overhead of maintaining and checking type information at runtime. Once a service has been validated, it is translated into C++ and compiled to

native shared-object code files. These files are then dynamically linked into RAE, at which point they are on-line and ready for event processing.

Safety and Complexity

In addition to type checking, the declarative nature of SAL makes two other static sanity checks possible. These checks—also performed prior to translation, compilation, and installation—address the issues of safety and complexity, both of which have important performance implications:

- *Safety*. Because service authoring is a high-level, largely declarative procedure, the process of checking and translation can statically guarantee the safety of authored services. For example, it is not possible to author services with errors such as infinite loops, memory leaks, erroneous updates through misdirected pointers, or segmentation violations.
- *Complexity*. SAL provides a primitive set of programming constructs and a small, but rich, set of declarative data management constructs. This combination is attractive for real-time systems, because the complexity of authored services can be analyzed statically. In many cases, SAL can give complexity guarantees and generate warnings if an authored service is potentially inappropriate for real-time event processing.

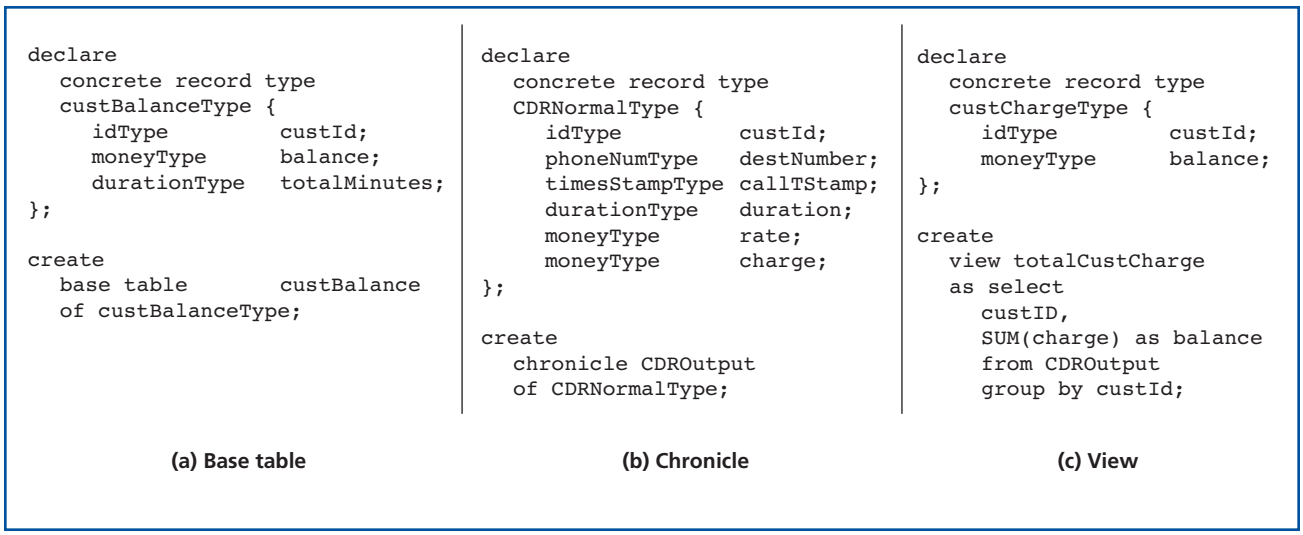
In the case of safety, these features improve performance by relieving RAE of the burden of addressing safety issues at runtime. Given the inherent correctness of the checking, translation, and compilation process, only a limited, well-defined and manageable set of error conditions (such as division by zero) can occur at runtime. As a result, authored code can be compiled and executed safely directly within the process space of RAE. If these guarantees could not be given, then dealing with runtime errors would require dynamic mechanisms. For example, authored services might be executed in a separate address space, as is done with client-server systems. This latter alternative, however, introduces considerable initialization, context switching, and data marshaling overheads, all of which can be avoided here. Ensuring the safety of authored code in this way, however, does not in itself ensure the correctness of that code. In particular, none of the features described here eliminates the need for conventional testing and debugging.

In the case of complexity, one key performance metric for real-time systems is predictability. The declarative nature of SAL makes it possible to distinguish code whose complexity is constant or logarithmic from code whose complexity is proportional to the size of the database.[11,12] Because this latter case may compromise predictability and real-time performance, it is flagged with a warning message, but not rejected. This gives an author the opportunity to reconsider the design of a service from a real-time perspective, eliminating a potential performance problem before it is introduced. Nevertheless, complex event processing can on occasion be unavoidable or appropriate. This might be the case, for example, if an event occurs only infrequently, or if the system's dynamics ensure that performance will in fact be satisfactory in practice (for example, a scan of a table may be reasonable if the table has only a few entries), or if the task at hand is inherently complex. In these cases, an author might in fact validate code that, on the face of it, appears inappropriate for real-time processing. For these reasons, event-processing code is never rejected automatically on the grounds of complexity alone.

Authoring Tables

The primitive data structure provided by SAL is the *table*, which comes in three types—base tables, views, and chronicles.

- *Base tables*. A base table is a regular, full-function table. The update operators for base tables are `insert`, `update`, and `delete`, and the query operator is `select`.
- *Views*. A view is a table whose contents are derived from the contents of other tables. Views have no update operators, only the query operator `select`.
- *Chronicles*. Chronicles are not stored within Sunrise; instead, an insertion into a chronicle generates an output from the system. The only update operator for chronicles is `insert`, and the query operator `select` may appear over chronicles only in view definitions.



```
declare
   concrete record type
   custBalanceType {
      idType          custId;
      moneyType       balance;
      durationType    totalMinutes;
};

create
   base table        custBalance
   of custBalanceType;
```

(a) Base table

```
declare
   concrete record type
   CDRNormalType {
      idType          custId;
      phoneNumType    destNumber;
      timesStampType  callTStamp;
      durationType    duration;
      moneyType       rate;
      moneyType       charge;
};

create
   chronicle CDROutput
   of CDRNormalType;
```

(b) Chronicle

```
declare
   concrete record type
   custChargeType {
      idType          custId;
      moneyType       balance;
};

create
   view totalCustCharge
   as select
      custID,
      SUM(charge) as balance
      from CDROutput
      group by custId;
```

(c) View

Figure 5.
Simple table-definition examples.

Figure 5 illustrates simple examples of these three forms of table, which are loosely based on Example 1, debit-based billing, shown earlier. The base table, `custBalance` (Figure 5a), might contain each customer's balance and usage. The chronicle, `CDROutput` (Figure 5b), records all the events processed by the system. The view table, `totalCustCharge` (Figure 5c), maintains aggregation information over `CDROutput`. It contains the total charges assigned to each customer, keeping these up to date automatically as outputs are generated.

Authored services explicitly maintain the contents of base tables. For example, transaction processing may insert, delete, or update a record within a base table. Generally, base tables store either configuration data or summary data, which cannot be maintained automatically as a view. A view is a table for which no update operators are permitted; instead, the contents of a view are derived from the contents of other tables (frequently from the contents of chronicles). For performance reasons, views are always stored explicitly in the memory store. Their contents are updated automatically as a side effect of transactions' updates to base tables and chronicles. Whenever possible (and always for views over chronicles), Sunrise uses efficient algorithms for incremental view maintenance.[3,11]

Base tables and views are stored explicitly in the memory store. Chronicles, on the other hand, are not stored within Sunrise. A chronicle models a stream of processed-event records. Inserting data into a chronicle is synonymous with generating a record on an output stream. A view over an output stream aggregates summary information over all the events processed by the system.[11] Frequently, chronicles represent either query results or processed-event records that are piped to a data warehouse for archiving. With respect to complexity, because an entire chronicle is not available for view maintenance, it is essential that views over chronicles be maintained incrementally, one record at a time.[11,12] In addition, views over chronicles must incorporate a "`group-by`" clause that bounds the space required for the view's materialization.

Authoring Transaction-Processing Logic

The SAL offers a primitive set of programming constructs and a small, but rich, set of declarative data-management constructs. The primitive programming constructs include assignment, sequencing, and conditional execution, but not loops. The data-management constructs are the four table operators of SQL (`insert`, `delete`, `update`, and `select`).

The `insert`, `delete`, `update`, and `select` operators are restricted versions of their SQL equivalents.[4] In particular, each operator accesses exactly one table in its `from` clause, and table names and table expressions may not occur elsewhere. With these restrictions, operations frequently correspond to rifle-shot dips into the memory store, which—with index

```
create service steppedPrice {
   declare
      concrete record type rateTableType {
         durationType  lowVol;
         durationType  highVol;
         rateType      rate;
   };

   declare base table rateTable
      of rateTableType;

   handle callSetup {
      .
      .
      .
   };
```

```
   handle callCompletion {
      durationType  useg;
      rateType      rate;
      set useg = pick from
                 select  totalMinutes
                 from    custBalance
                 where   custId =  cdr.custId;
      set rate = pick from
                 select  rate
                 from    rateTable
                 where   lowVol < useg
                         and useg <= highVol;
      set cdr.charge =  cdr.charge +
                        rate * cdr.duration;
      set cdr.rate   =  cdr.rate * rate;
   };
};
```

Figure 6.
A simplified `steppedPrice` ***service.***

support—can be executed in constant or logarithmic time. As such, these operators are well suited to real-time processing.

Consider, for instance, a (simplified) stepped pricing plan whose rates vary according to the current volume of usage a customer has generated. **Figure 6** illustrates the logic for this plan's `callCompletion` event handler. First, the customer summary information is accessed to determine the customer's usage, and, based on that usage, a rate table is accessed to determine the rate for a call. Finally, the rate is applied to the call detail record (CDR) at hand. In general, a `select` operation gathers a set of records. The "`pick from`" operator extracts a single member from that set (in this case it must be a singleton set). As this example illustrates, although restrictions are imposed on the data-management operators, combining these simple operations can render relatively sophisticated functionality. In the example above, processing that could be expressed by joining the `custBalance` and `rateTable` tables is instead expressed in terms of two rifle-shot `select` operations.

Services

The authoring model is based on the idea of services. A service provides a coherent set of event handlers that, together, implement some unit of functionality. For instance, in Example 1 (the debit-based billing example), handlers for `callSetup` and `callCompletion` events generally work together in pairs, with each pair grouped into a single service. **Figure 7** presents a set of services that might apply in a debit-based billing system. An 'O' in this matrix indicates that the service on the *x*-axis provides a handler for the corresponding event on the *y*-axis. An 'X' indicates that no handler is provided. For example, the `steppedPrice` service provides handlers only for `callSetup` and `callCompletion` (as illustrated also in Figure 6), whereas the `volumeDiscount` service (which maintains per-subscriber summary information) also provides handlers for `subscribeCustomer` and `unsubscribeCustomer`.

The SAL provides a set of features for defining services, events, and event handlers. Event handlers are coded using the features described in the previous section. Figure 6 shows examples of a service and an event-handler definition. In addition, associated with each event is a method for determining the subscriber for the event and the time at which the event takes place (if that time is not simply "now"). This information is necessary for the subscription model described below. In addition, a new service can inherit behavior from an existing service, redefining only those handlers whose functionality differs from that of the existing service.

Subscription Model

Generally, database systems support two classes of queries: *ad hoc queries* and *canned queries*. Ad hoc queries offer flexibility, but they incur considerable overhead at

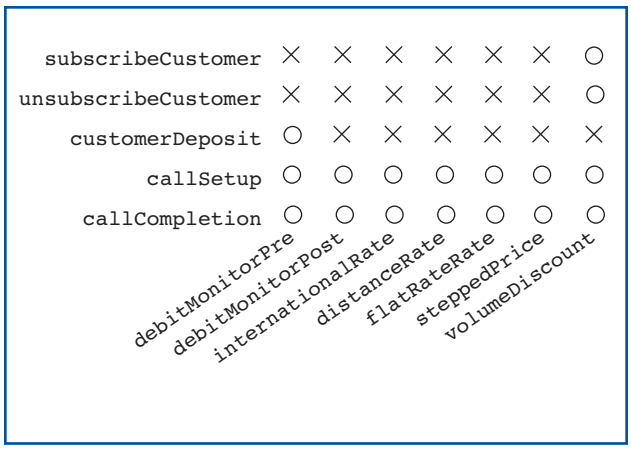


Figure 7.
A sample service/handler matrix.

runtime from parsing, planning, and optimizing costs. Canned queries, on the other hand, eliminate the overhead of ad hoc queries at the expense of flexibility. Sunrise strikes a balance between these two extremes by using a subscription model, which minimizes the overhead of event processing, while retaining much of the flexibility of ad hoc query processing systems.

The subscription model works as follows. A compact subscription table is maintained, as illustrated in **Figure 8**. Given an event with subscriber '`sub`' and time stamp '`ts`', this table contains a unique entry '`S`' such that:

```
S.subscriberId == sub and
S.startTimeStamp <= ts and
ts < S.endTimeStamp
```

The explicit subscription term for an event is then given by '`S.subscription`'. With appropriate index support, this term can be accessed in constant time. Two special identifiers, '`preProcessingSub`' and '`postProcessingSub`', also have entries in this table. In addition to explicit subscription terms, these entries represent implicit subscription terms to which all subscribers subscribe.

The *complete subscription term* for an event is derived by concatenating the preprocessing subscription term, the explicit subscription term, and the postprocessing subscription term, in that order. Moreover, the pre- and postprocessing terms can be cached between events, so they do not in fact require a lookup operation. For example, if an event for `Frank` occurs during August 1997, then the complete subscription term for the event (schematically illustrated) would be:

```
——d——; ——b——; ——e——
```

This term then defines the services whose handlers are used to process the event. The key advantage of this approach is that a single table lookup accesses all the subscription information for an event. Subscriptions themselves are encoded in a small but flexible subscription language. For example:

```
flatRateRate;
volumeDiscount;
```

service should be invoked first, followed by the handler of the `volumeDiscount` service. Terms in this language are then compiled to a very compact form (such as represented here schematically as '`——a——`') ideally suited to efficient run-time interpretation with minimal CPU and space overheads. This compact approach can be contrasted with existing subscription-based systems in which subscription information spans several tables, or even several entries within each table. As such, at runtime each event has considerable data access and interpretation overhead, almost all of which is avoided here.

In practice, subscription itself can be complex. For example:

- Services may be active only for a particular time period,
- Ordering dependencies may exist between subscriptions,
- Some services (such as those implementing taxation) are subscribed to implicitly by all subscribers,
- An event may need to "qualify" for a service, or
- "Preclusion" or "prerequisite" dependencies may exist between services.

The subscription language is sufficiently rich to capture these and a variety of other classes of dependency, not only between services, but also between subscriptions.

Authoring Input and Output Streams

In Sunrise, many target applications serve as adjunct services to embedded systems, such as network switches. To exchange data with such embedded systems, Sunrise supports stream-based external interfaces. An input or output stream is a sequence

subscriberId	startTimeStamp	endTimeStamp	subscription
Frank	1/1/97	3/31/97	a
Frank	4/1/97	12/31/97	b
Steve	6/30/97	6/30/98	c
preprocessingSub	1/1/97	12/31/98	d
postProcessingSub	1/1/97	12/31/98	e

Figure 8.
A sample compact subscription table.

of records, generally (but not always) transmitted over a transmission control protocol/Internet protocol (TCP/IP) connection. Authoring streams involves describing the physical record formats that can occur on a stream; identifying key features, such as format descriptors and length indicators, embedded within those records; and defining mappings between record formats.

When Sunrise interfaces with existing systems, two complications arise. First, stream formats generally conform to existing standards, so Sunrise must be capable of interfacing with systems that use those standards. Second, streams are frequently not homogeneous, but rather comprise a variety of different record formats on a single stream. The SAL includes several features for addressing these issues, one of which is the ability to perform normalization over multiformat input streams. Even if a stream contains several input formats, they can be consolidated into a single internal format on arrival, and only this uniform internal format is processed within Sunrise.

For instance, consider Example 2, the fraud-detection example shown earlier, based on processing streams of CDRs. About fifty different CDR formats arise in practice. If the authoring process is independent of these individual formats, it can be simplified by authoring each physical record format, superimposing a normalized record format over the physical record format, and providing a normalization for each CDR format.

Figure 9 illustrates this process for two CDR formats—`exampleCDRType1` and `exampleCDRType2`. Services are authored not in terms of either of these specific formats directly, but rather in terms of the third, normalized format, `CDRNormalType`. A normalization is declared for both of the CDR formats, and these normalizations map CDRs to the normalized form for processing.

Although this approach simplifies authoring itself, normalization is "lossy." In the example above, the `sensor` field of `exampleCDRType1` does not occur in the normalized form and is therefore discarded by the normalization process. As a consequence, this approach is satisfactory only if downstream processing does not rely on a record's specific input format. In practice, this precondition cannot be taken for granted.

To overcome this difficulty, Sunrise provides a second form of normalization, termed *abstract normalization*, based on the concept of an abstract record type. Abstract record types are similar to concrete record types, but they define an interface to a group of record formats rather than a physical record format. In this case, mappings between these concrete and abstract record types enable a single interface to be used to process records that have a variety of different underlying physical formats, without requiring normalization to be performed. For example, if the declaration of the type `CDRNormalType` in Figure 9 were given not as a concrete type, but rather as an abstract type:

```
declare concrete record type
  CDRNormalBaseType {
    dollarCentType rate;
    dollarCentType charge;
};
declare abstract record type CDRNormalType
    under CDRNormalBaseType {
        idType custId;
        phoneNumberType destinationNumber;
        timeStampType callTimestamp;
        durationType duration;
};
```

then normalization is abstract. In this case, if the

```
declare concrete record type
   exampleCDRType1 {
      intType          format;
      idType           custId;
      phoneNumberType  destinationNumber;
      durationType     duration;
      dateType         calldate;
      timeType         callTime;
      idType           sensor;
} with key format == 0x0001;

declare concrete record type
   exampleCDRType2 {
      intType          format;
      idType           custId;
      dateType         callDate;
      timeType         callTime;
      intType          hours;
      intType          minutes;
      phoneNumberType  destinationNumber;
} with key format == 0x0002;
```

```
declare
   concrete record type CDRNormalType {
      idType           custId;
      phoneNumberType  destinationNumber;
      timestampType    callTimestamp;
      durationType     duration;
      dollarCentType   rate;
      dollarCentType   charge;
};

create normalization of exampleCDRType1
   as CDRNormalType with {
      callTimestamp    = mkTS(callDate,callTime);
};

create normalization of exampleCDRType2
   as CDRNormalType {
      duration         = minutes + 60 * hours
      callTimestamp    = mkTS(callDate,callTime);
};
```

Figure 9.
Input stream authoring for a simplified CDR stream with two different record formats.

underlying record is of type `exampleCDRType1`, then a reference to the abstract field `duration` is replaced with a reference to the concrete field `duration`. If the underlying record is of type `exampleCDRType2`, it is replaced with the expression `minutes + 60 * hours`. By defining transactions in terms of abstract records, multiformat inputs are mapped to multiformat outputs without loss of information.

Conclusion

This paper described Sunrise, a real-time event-processing system conceived at Bell Labs. While conventional database systems provide support for high-throughput applications, many new applications are emerging that require not just high throughput, but also real-time responsiveness. Sunrise has been designed to meet the requirements of these real-time applications. Such applications are common in telecommunications networks for debit-based billing, fraud detection, call centers, hot billing, and adjunct switching services such as local-number portability and toll-free number mapping. Sunrise meets real-time performance goals by using the DataBlitz main-memory storage manager as its underlying database system. The DataBlitz storage manager offers transactional access to persistent data at the speed of main-memory systems. Moreover, Sunrise uses a shared-nothing approach to parallelism that can scale well as workload and resources increase.

Sunrise incorporates an authoring tool kit—the SAE—to develop application-specific services. The SAE provides a set of tools and GUIs for authoring, validating, compiling, and installing new services or existing services adapted to new requirements. The SAL provides features for defining input and output streams, event-processing logic, the summary and aggregation tables necessary to support that logic, and the canned queries supported by a system. This paper also described a novel subscription model—the basis for flexible, low-overhead, real-time event processing—and sketched its implementation.

Performance tests were conducted based on a simple, update-intensive rating and discounting application. Run on a SUN Microsystems workstation with a single UltraSPARC* processor, these tests demonstrated a throughput of about 600 events per second. As these tests were being conducted, a prototype data warehousing system based on the Oracle system was running on the same CPU. Separate performance tests using the DataBlitz storage manager directly have shown transaction-processing rates of about 860 updates per second and nearly 60,000 reads per second. These results clearly demonstrate the performance capabilities of a well-engineered, general-

purpose event-processing system based on the DataBlitz main-memory storage manager platform.

Acknowledgments

Many people have contributed to the Sunrise project. Particular recognition is due to Yuri Breitbart, Leonid Libkin, and Steve Buroff of Bell Labs Research, all of whom contributed to the design of the SAE; to all the members of the Sunrise development team based in Columbus, Ohio; Cary, North Carolina; and Murray Hill, New Jersey; and to the DataBlitz team, also part of Bell Labs in Murray Hill.

*Trademarks

"Friends and Family" is a registered trademark of MCI Communications Corporation.

Java is a trademark of Sun Microsystems, Inc.

Oracle is a registered trademark of Oracle Corporation.

SPARC and UltraSPARC are registered trademarks of SPARC International.

"The Most" is a registered trademark of Sprint Communications Company L.P.

References

1. Igor Faynberg, Lawrence R. Gabuzda, Marc P. Kaplan, and Nitin J. Shah, *The Intelligent Network Standards*, McGraw-Hill, New York, 1997.
2. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, Reading, Mass., 1987.
3. Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, Inc., San Francisco, 1993.
4. Abraham Silberschatz, Henry F. Korth, and S. Sudarshan, *Database System Concepts*, 3rd ed., McGraw-Hill, New York, 1997.
5. Philip L. Bohannon, Rajeev R. Rastogi, Avi Silberschatz, and S. Sudarshan, "The Architecture of the Dalí Main Memory Storage Manager," *Bell Labs Tech. J.*, Vol. 2, No. 1, Winter 1997, pp. 36–47.
6. H. V. Jagadish, Daniel Liewen, Rajeev Rastogi, Avi Silberschatz, and S. Sudarshan, "Dalí: A high performance main memory storage manager," *Proc. Intl. Conf. on Very Large Databases (VLDB)*, Santiago, Chile, Sept. 1994, pp. 48–59.
7. David DeWitt, Randy Katz, Frank Olken, Leonard Shapiro, Michael Stonebraker, and David Wood, "Implementation techniques for main memory database systems," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Boston, Mass., June 1984, pp. 1–8.
8. T. J. Lehman and M. J. Carey, "A study of index structures for main-memory database management systems," *Proc. Intl. Conf. on Very Large Databases (VLDB)*, Kyoto, Japan, Aug. 1986, pp. 294–303.
9. T. Lehman, E. Shekita, and L. Cabera, "An evaluation of Starburst's memory-resident storage component," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 4, No. 6, Dec. 1992, pp. 555–566.
10. K. Salem and H. García-Molina, "System M: A transaction processing testbed for memory resident data," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 2, No. 1, Mar. 1990, pp. 161–172.
11. H. V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz, "View maintenance issues for the chronicle data model," *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS)*, San Jose, Calif., May 1995, pp. 113–124.
12. Timothy Griffin and Leonid Libkin, "Incremental maintenance of views with duplicates," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Jose, Calif., May 1995, pp. 328–339.

(Manuscript approved October 1997)

GERALD D. BAULIER is head of the Systems Modeling Research Group at Bell Labs in Murray Hill, New Jersey, where he is responsible for main-memory database systems, real-time event-processing billing systems, and fraud systems. Mr. Baulier received a B.S. from the University of Massachusetts—North Dartmouth and an M.S. from the Stevens Institute of Technology, Hoboken, New Jersey, both in computer science.

STEPHEN M. BLOTT earned B.S. and Ph.D. degrees in computing science from the University of Glasgow in Scotland. As a member of technical staff in the Database Principles Research Department at Bell Labs in Murray Hill, New Jersey, Dr. Blott conducts research on database systems, transaction processing, real-time systems, and telecommunication systems.

HENRY F. KORTH holds a B.S. in mathematics from Williams College in Williamstown, Massachusetts, and M.S.E., M.A., and Ph.D. degrees in computer science from Princeton University in New Jersey. Dr. Korth, head of the Database Principles Research Department at Bell Labs in Murray Hill, New Jersey, is responsible for high-performance database systems, transaction processing, data replication, billing systems, and data warehousing.

AVI SILBERSCHATZ is director of the Information

Sciences Research Center at Bell Labs in Murray Hill, New Jersey. He holds M.S. and Ph.D. degrees in computer science from the State University of New York at Stony Brook. Dr. Silberschatz directs work on operating systems and high-performance databases. ◆