SUNRISE II: A Scalable and Timely Billing Architecture

H. V. Jagadish*
AT&T Laboratories
jag@research.att.com

Inderpal Singh Mumick*
AT&T Laboratories
mumick@research.att.com

Abstract

SUNRISE II is an architecture describing the management of customer and usage data for billing and other account services in any transaction based system. The architecture provides real-time rating, discounting, and information services based on the chronicle data model without using any specialized database software. Further, the architecture allows for a connection between the billing data stream and network management data, quick creation of new features, automatic detection of interactions between features, a model of using self-maintainable views to integrate customer data coming from multiple sources, and techniques to store large volumes of billed transactional data in an on-line system.

1 Introduction

Billing plays a major role in communication *services* provided over a network. Rating and billing information, if available in real-time, can enable new rating and information services, as well as other useful non-service functionality, such as fraud control. A flexible billing system that permits quick introduction of new features can result in a competitive advantage.

The SUNRISE II system defines an architecture for billing usage data, such as telephone calls and internet transactions, as soon as they are recorded, and for providing information services based upon real-time access to a summary of the usage and rating information. For example, the query "How many dollars have I spent on international telephone calls this month?" can be answered with a delay measured in milliseconds. This delay is small enough that such queries can be asked during call setup, allowing the billing data stream to influence the management of the network. For instance, such a query can be used to detect potential fraud in real time, and prevent it by call denial. The SUNRISE II architecture is based on an earlier architecture [JKMS94] that used specialised database software to do billing.

There are several problems that must be solved in order to build such a system. First, the sheer volume of customer and transactional data can make it difficult to store, rate, and query the data. Secondly, using the billing data stream for network services requires queries to be answered in small fractions of a second, which does not give us time to retrieve data from the disk; so such data must be stored in memory. Third, the number and complexity of rating, discounting, and information services is growing, and the interactions between these services can slow down the development of new features. In fact, most current systems have a turn-around time of several weeks, or even months for introduction of new features. Fourth, the customer data needed to process the billing

*This paper describes a futuristic billing system. It does not reflect the views or plans of AT&T or any other corporation.

data stream is usually not available at one place, and we need to figure out a way of combining these data sets without taking over ownership of the data sets. Fifth, the billing plans are typically defined as batch plans that need the full month's data before they can be applied. How do we apply them incrementally as data for each transaction is generated?

Real-Time Pricing: The SUNRISE II system identifies a two layer database architecture - a relational database store for customer and summary data, and a main memory store of essential customer and summary data. While the former is stored in a commercial database, the later is managed by a non-database application software. The SUNRISE II system processes transactional data as it is produced. Processing involves all billing activity, as well as accumulation of summary information. The nature of the summary information depends upon the features subscribed to by the customer, and is crucial in being able to apply batch type billing plans in real-time. For instance, for a volume discount plan, we accumulate summary of charges for all usage until the current instant. As the accumulated charges grow, we keep on increasing the volume discount offered, and adjusting the accumulated charges based upon the current discount percentage. The essential summaries are stored in memory, can be queried very fast, and can be used to control the network on which the transactions are executing. For instance, in a telephone network, queries can be posed against the accumulated summary information during call set-up, and such queries will be answered in real-time. A technique to do real-time rating using specialized database software was described earlier in an internal technical report [JKMS94].

Detailed Data: SUNRISE II uses a distributed architecture for storing the detailed transactional data after it is billed. Billing information about each transactional record is stored inside the data record. The data is fragmented amongst multiple servers to handle the large volumes and transactional rates. We show that fragmenting the data by customers can cut down on the need for communication between the database servers. A benchmark of the detail storage system is described in [NOLC95].

Customer Profile: SUNRISE II uses an "integrated customer profile" that can be generated from component customer databases owned by different organizations. The customer profile is defined as a self-maintainable view, and is materialized within the SUNRISE II system. Materialized views [Mum95] are a tool to store integrated customer data while allowing updates to continue to occur on the component systems that own the data. View Maintenance [GM95] techniques are used to propagate changes to customer data onto the integrated customer profile. A view is said to be self-maintainable [GJM96] if it can be maintained easily, without communication overhead. We describe how to use self-maintainability in defining and maintaining the integrated customer profile, thereby making it's use possible in a high performance transactional system.

Quick Introduction of discount plans: SUNRISE II uses a meta-rule language to specify resolution of conflicts between different rating and discount plans. For instance, one can specify that only one of two volume discount plans can apply to a usage record, or that two discounts are to be applied in a particular order. The meta-rules can then be analyzed automatically to determine whether a unique execution of the discount plans is possible for every customer. Details of the meta-rules and their model theory is described in [JMM96]. These techniques are used in SUNRISE II to speed up the definition and introduction of new discount plans.

Data Model: SUNRISE II uses the chronicle data model [JMS95] to define the incremental computation that achieves the effect of batch style billing plans.

Scalability: The SUNRISE II architecture is scalable, and can grow with increases in transactional volume, number of customers, and number of features.

New Services: Several new services are possible using the SUNRISE II architecture. As one example, we can do hot billing, wherein a customer can be sent a bill on demand. As another example, we can bill at any frequency we desire, triggered either by time or by a dollar expense.

Paper Outline: The remainder of this paper is organized as follows. We first introduce the SUNRISE II architecture in Section 2, and then discuss each of the major components of the architecture — Real-time pricing (Section 3), the call detail database (Section 4), the integrated customer profile (Section 5), and the service creation platform (Section 6). Conclusions are given in Section 7.

2 SUNRISE II Architecture

The SUNRISE II architecture, outlined in Figure 1. contains four major components:

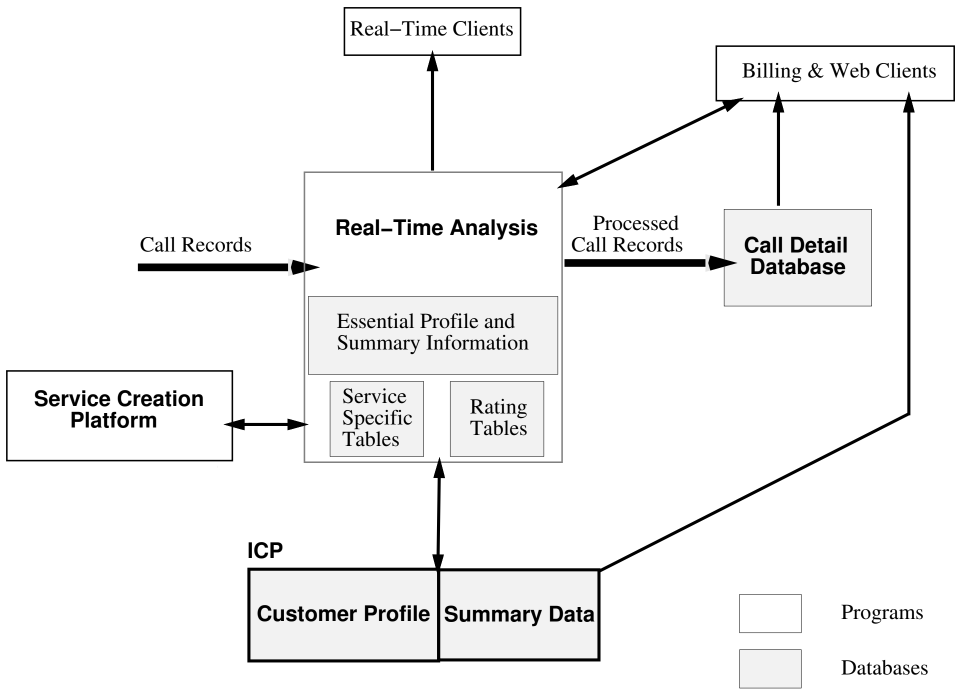

Figure 1: The SUNRISE II Architecture.

- The Real-time Analysis Engine (RAE).
- The Call Detail Database (CDD).

- The Service Creation Platform (SCP).
- The Integrated Customer Profile (ICP)

Real-time Analysis Engine The Real-time Analysis Engine (RAE) is at the heart of the architecture. The RAE server processes the incoming stream of call detail records, and for each call detail record, (1) rates and discounts the call detail record according to *customized* rating and discounting functions that depend on the list of features and services subscribed to by the customer, (2) computes summary fields associated with the customer, (3) generates and compresses a processed call detail record that includes the rating and discounting information, and (4) stores an in-memory image of summary fields and customer information essential for rating, discounting, and real-time queries. Additionally, if needed, the RAE server responds to queries about customer profile and summary data.

Call Detail Database The Call Detail Database stores processed call detail records. The processed call detaul records contain rating and discounting information, and any information relevant for computing summaries. The call detail database services several clients that need the call detail information. For example, bill rendering is done by simple client programs that access processed call detail for the group of customers that need to be billed, format the records, and generate a printable image of the bill. These client programs also have access to the essential summaries maintained by the RAE server, and customer profile maintained at the ICP. The bill rendering clients do not need to do any rating or discounting. Marketing and decision analysis programs can also run as clients on the call detail database. The RAE process also accesses the call detail in case it needs to re-rate or re-discount calls, or needs to recover from a failure.

Service Creation Platform The Service Creation Platform helps in creating new features. A feature can be a rating plan, a discounting plan, a promotion, an information gathering request by the customer or by an internal organization. The service creation platform consists of a graphical interface to specify a new feature, a mechanism to translate the specifications into definitions of new summary fields, and functions to compute the summary fields. Further a sophisticated technique to resolve conflicts between different features (e.g, conflicts between two discount plans) is available as a part of the service creation platform.

Integrated Customer Profile The integrated customer profile (ICP) represents a logical integration of service specific customer profiles maintained by distinct service providers. In addition, summary fields can be defined for each customer and stored alongside the customer profile. Summary fields summarize data from all the call records billed to the customer. The summary data is sent to the ICP from the RAE at specified periods. Summary fields are defined by service providers for the customers who subscribe to individual services.

Key Features: The above four components support the following key features in the SUNRISE II architecture:

- A large number of rating and discounting features can be supported, including cross-service discounting for convergent billing.
- Triggers can be activated upon observing *interesting* sequences of call detail records (*e.g.*, for Fraud Detection).

- Real-time querying over up-to-date summaries is possible, even as a part of call set-up. Such queries can be answered in about 1 millisecond.
- An integrated view of the customer profile is maintained. Existing service providers can continue to retain control over their own data. New service providers may choose to build their own database, or to keep their data in the integrated customer profile. In either case, the data needed for offering new services and billing is accessible as a logical database.
- Several bill rendering and collection options (with and without call detail) are available. Billing without call detail can be done at any frequency, with low overhead. It is possible to generate daily bills, for example, and use this to debit an account electronically.
- New discounting and information collecting features can be created rapidly and efficiently.
- At the time of feature creation, assistance is available in determining feature interactions.
- The architecture is scalable with respect to increases in number of call detail records, increases in number of features offered, and increases in number of subscribers.
- It is possible to introduce the SUNRISE II architecture into an existing telephone network gradually.

Key Technologies: The proposed SUNRISE II architecture is enabled by three key technologies.

Chronicle Data Model: The chronicle data model [JMS95] provides for real-time computation of summaries over streams of data. The call data represents a data stream, or chronicle. Each customer is allocated storage space that depends upon the number of features subscribed, and is independent of the number of call detail records billed to the customer. The summary fields are defined for each feature at time of feature creation, and can be periodically reset (*e.g.*, to initiate new billing cycles). No storage space is allocated for features that a customer does not subscribe.

Incremental View Maintenance: The integrated customer profile is derived as a materialized view from multiple service specific customer profiles maintained by each service provider. As the underlying service specific data changes, the consequent changes to the integrated customer profile are computed incrementally, without recomputing the entire integrated customer profile. Similarly, the summaries are defined as a view over the call detail and ICP, and are maintained incrementally as each new call detail record is processed. Incremental view maintenance techniques to help maintain the materialized views are discussed in [GM95, MQM97, GJM97].

Conflict Detection: Various discounting plans can conflict with each other. A plan may require that another plan be disabled, or that another plan be subscribed. The order in which two plans are applied can influence the total discount. A set of meta-rules can be used to specify such conflicts between discounting rules [JMM96]. The meta-rules can be reasoned with to derive the set and order of discounting rules that should apply to a customer.

3 Real-Time Pricing

We first distinguish between *rating*, *discounting*, *promotions*, *pricing*, and *billing*. We then introduce the concept of real-time pricing, and show how it might be supported using summary information. Finally, we present the RAE architecture for doing real-time pricing.

3.1 Rating, Discounting, Pricing, and Billing

Rating is the process of assigning a rate to a call, with consideration for subscribed `rating` plans. Rating thus depends upon (1) Service independent rating tables that take into account the time of day, the day of week, distance, and length of call, and (2) List of services subscribed to by the accounting entity, and (3) Service dependent rating tables that determine how each service affects the rates for a call.

Discounting is the process of computing discounts on a call, with consideration for subscribed `discounting` plans. Discounting occurs after rating, and usually computes a credit over the original rate assigned to the call. Multiple discounts can apply to a call. Two different discounts can both apply to the original rate, or one can apply to the discounted rate.

Promotions are short term incentives given to customers. They are like discounting plans, but they only apply for a short term. Promotions may apply on the original non-discounted rates, or on discounted rates.

Pricing is the process of computing a price for a call by doing rating, discounting, and applying promotions.

Billing is the process of *aggregating over the rate charged for each call* made in some period, usually a month, to determine the final aggregate amount to be charged to the customer, followed by generation of a printable image of the bill. Billing thus consists of (1) pricing, (2) end of period processing (aggregating the priced calls, applying monthly fees and credits), and (3) formatting a bill. Most of the complexity of billing is really in the pricing component. If calls are priced adequately, the end of period processing and bill rendering can be done easily by small customized, clients.

The SUNRISE II architecture separates the pricing functionality into the RAE server, and requires bill rendering clients to do the end of period processing and formatting. Further, the pricing is done by RAE in real-time, while the bill rendering clients can be run at any periodic frequency to generate the bills.

3.2 What is Real-Time Pricing?

Real-time pricing means that the exact cost of a call be computed as soon as the call detail record is fed into the RAE server, even if the call detail record comes in immediately after the call is completed. This may not be possible always, especially since discounting plans are defined to apply over a set of call detail records for a customer. However for most discounting plans, it is possible to translate the set-oriented discounting into an incremental call at a time real-time discounting, as the following example illustrates.

EXAMPLE 3.1 Consider a volume discount plan where a 10% discount is applied if more than 15 hours of calls were made in one month. The 10% discount is given on all *subsequent* calls after the first 15 hours of calls. Such a discount can easily be computed in a real-time incremental fashion. As each call comes in, a running summary of the total length of calls made by a customer is maintained in memory. The call is discounted at 0% if the total length summary has a value less than 15 hours, and at 10% if the total length summary has a value greater than or equal to 15 hours.

Now consider a slight variation where the 10% discount is to be applied to *all* calls, including the calls made within the first 15 hours. In this case, when processing the first 15 hours of calls we do not know whether these calls will be eventually discounted. Thus the processed call detail records cannot reflect this discount in their priced value. However, calls processed subsequent to 15 hours of calls can reflect the correct discount. It may appear that real-time pricing fails !

However, we can still make real-time pricing work if we keep a summary of total price charged for all calls. The summary information about the total charges for the calls can be adjusted at the time it is noticed that the total usage has gone beyond 15 hours, by taking a 10% on the charge accumulated so far. Thus, even with this plan, "correct" summary information regarding total charges is accessible at all times. However, the actual price put on the processed call detail records for the first 15 hours of calls needs to be adjusted by applying a compensatory threshold discount. ■

As should be clear from the above example, summary data is crucial to real-time pricing. The summary data must be maintained at a customer level to do pricing. Other types of summary data might be maintained at higher levels to do journaling or to collect marketing information in real-time.

3.3 The RAE Server

The RAE server contains customer data and essential summaries needed to do real-time rating, discounting, and promotions. Further, since the call detail records are expected to be fed into the RAE server in a random order at a very fast pace, the customer information and summaries need to be accessed in a random order, at speeds that keep up with the input rate. Doing so requires that all the customer information and summaries needed for pricing be available in an in-memory buffer.

The RAE server consists of a cluster of identical machines working in parallel. Each RAE machine handles a horizontal partition of the call records, and a horizontal partition of the customer profile and summary data, partitioned by customer-id. A distribution manager distributes call detail records between the RAE machines using the same criteria used to distribute the ICP and summary data.

The main activity in each RAE machine is the call processing process that accepts a stream of call detail records as input, prices each record, does summary computation, and outputs a stream of processed call records. In addition, there are processes to respond to queries over the customer profile and summary data, to send a snapshot of the summaries into the ICP store, and to maintain the customer profile in response to provisioning changes in the ICP.

The call processing process treats the incoming stream of call detail records as a chronicle, and processes the summary computation according to the chronicle data model [JMS95]. We describe the chronicle data model next.

3.4 The Chronicle Data Model

The subsection describes results previously reported in [JMS95].

3.4.1 Model Definition

A chronicle database consists of *relations*, *chronicles*, and *persistent views*. *Relations* are standard, as in any relational database. Each relation may have several temporal versions (at least conceptually).

A *chronicle* is similar to a relation, except that a chronicle is a *sequence*, rather than an unordered set, of tuples. A chronicle can be represented by a relation with an extra *sequencing* attribute, whose values are drawn from an infinite ordered domain. The only update permissible to a chronicle is an insertion of tuples, with the sequence number of the inserted tuples being greater than any existing sequence number in the chronicle. There is no requirement that the sequence numbers be dense. Chronicles can be very large, and the entire chronicle may not be stored in the system.

There is a temporal instant (or chronon) associated with each sequence number. All operations on a tuple of a chronicle are with respect to the database as it was at that point in time. Thus, any join of a chronicle C and a relation R is a union of the corresponding joins of each tuple in C with the version of R that existed at the temporal instant of the tuple in C.

The database maintains a fixed number of *persistent views*, which are views that are materialized into relations, and are always maintained current in response to changes to the underlying database. Each persistent view is materialized when it is initially defined, and it is kept up-to-date, reflecting all the changes that occur in the database, as soon as these changes occur. Of particular concern is the maintenance of a persistent view after every append to a chronicle.

Persistent views are defined in a view definition language $\mathcal{L}$. The persistent views correspond to the procedurally computed summary fields in the current transaction systems. Each choice of $\mathcal{L}$ derives a particular instance of the chronicle data model. Formally,

Definition 3.1 (Chronicle database system): A chronicle database system is a quadruple:

$$(\mathcal{C}, \mathcal{R}, \mathcal{L}, \mathcal{V}),$$

where $\mathcal{C} = \{C_0, C_1, \ldots, C_n\}$ is a set of chronicles, $\mathcal{R} = \{R_0, R_1, \ldots, R_m\}$ is a set of relations, and $\mathcal{V} = \{V_0, V_1, \ldots, V_l\}$ is a set of persistent views defined in a language $\mathcal{L}$. □ ■

Example 2.1: Consider a billing system for tracking charges for cellular calls. There is one chronicle – the sequence of calls coming into the system. There is at least one relation containing information about customers, including their account number, name, and address. There are at least three persistent views to hold the total number of minutes, total basic charges, and total discounted charges of each customer. In order to define these persistent views, the language must allow for aggregation and joins between the chronicle and the relation. □

3.4.2 Queries

Queries that access the relations and persistent views can be written in any language — relational algebra, SQL, Datalog, etc. The choice of this language is orthogonal to the chronicle model. The chronicle model enables fast response to queries that access the persistent views; these queries may otherwise have been defined as complex SQL queries over the relations and chronicles and thus would not likely have been answerable with acceptable performance. Further, a system would typically provide detailed queries over some latest window on the chronicle.

3.4.3 Updates

There are two types of updates in our model: those that modify the relations and those that append to chronicles.

An update to a relation R can be an insert, delete, or modification of a tuple in R. An update to a chronicle C can only be the insertion of a new tuple (or tuples) to C with a sequence number greater than the sequence number of all existing tuples in C. We consider these updates in turn below.

Each relation conceptually has multiple temporal versions, one after every update. In any persistent view defined in language $\mathcal{L}$, any joins between the relations and chronicles have an implicit temporal join on the sequencing attribute: We can associate a temporal version of the relations with each sequence number in the chronicles. Each tuple of a chronicle is then joined with the version of the relations associated with the same sequence number.

If an update to a relation affects only the versions corresponding to sequence numbers not seen as yet, then it is a *proactive* update; such an update does not affect the persistent views. Only subsequent chronicle updates see the new relation values. Since maintainability of persistent views is critical in the chronicle model, we have chosen to limit the language $\mathcal{L}$ so that only proactive updates to relations are allowed.

In contrast, a *retroactive* update to a relation would require older tuples in the chronicle to be re-processed. Such updates, when necessary, are computationally expensive to maintain, may not be maintainable if the entire past chronicle is unavailable, and are not included as part of the chronicle model.

Example 2.2: Consider again the billing system example. Suppose that each customer living in New Jersey gets the first minute free on every call. The customer relation is updated whenever a customer changes his/her address. A call tuple in the chronicle qualifies for the free minute only if the call was made during the period of residence in New Jersey. Thus, the join between the chronicle and the relation is based on the temporal version of the relation associated with the sequence number in the chronicle. An update to the relation is proactive if the address update occurs before the associated call tuples are appended to the chronicle. □

An update to a chronicle may cause a change in each persistent view. The maintenance of persistent views is discussed in [JMS95],

3.5 Limiting the size of customer profile and summary data in memory

The RAE server stores parts of customer profile and summary data that are essential for real-time rating in main-memory in order to get the throughputs and response times needed for call processing and queries to the RAE. We must therefore ensure that the amount of profile and summary data needed by the RAE is not allowed to grow in an unbounded fashion.

We use two techniques to manage the storage requirements. First, a customer's record consumes storage only for the features used by the customer. Features are used by a customer if the customer explicitly subscribes to the feature, or if a feature is a "global" feature available to all customers. Global features are kept to a minimum.

Second, following on the chronicle data model, we only permit a fixed pre-determined amount of storage per feature. In other words any discount plan that requires an unlimited number of summaries may not be priced by the RAE server. For example, a discount plan that gives an extra 10% discount on the tenth call made to a number x, for every number x called by a customer, could potentially need an unlimited number of summaries, and cannot always be supported by the RAE server. For most customers, the amount of storage required may be small enough that the RAE can handle this. For a few customer who call more numbers frequently than the storage in the RAE can keep track of, such a discount plan could be applied by the billing programs that work off the processed call-detail in the call-detail database.

4 Call Detail Database

4.1 The CDD Architecture

Figure 2 shows the architecture of the call detail database (CDD) subsystem of the SUNRISE II architecture.

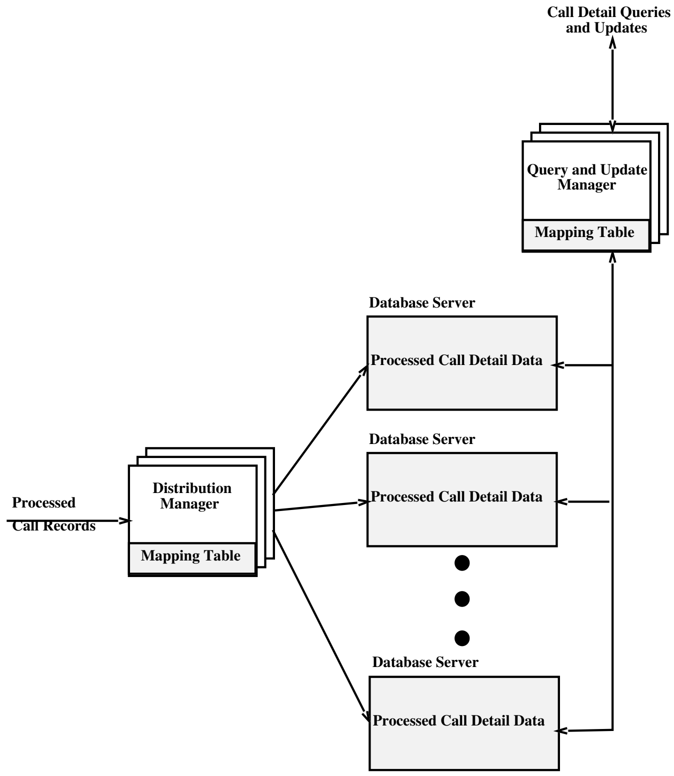

Figure 2: The SUNRISE II Call Detail Database Architecture.

The processed call detail data is horizontally fragmented across a bank of database servers. This partitioning is performed in a manner such that there is no significant interaction of data lying on different servers. For instance, joins between data lying on different servers is considered impractical. The input records can be partitioned to a database server depending on the customer-id of the customer. A map between the customer-ids and the database servers is stored in a mapping table. Each database server holds all the processed call detail records for a set of customer-ids assigned to the database server. All the records for one customer must belong to a single database server. This means that each database server must be big enough to hold the all records of the largest customer. Beyond the above size constraint, there is no other constraint, imposed by the CDD architecture, on the size of a database server. However, in a practical installation, we expect each database server to be as large a system as can be profitably built, so that the total number of database servers is small. This architecture is optimized to (1) access detailed records for a single customer, or for a set of customers, and (2) insert new processed call detail records for customers, arriving in a random order at a very high rate, and (3) update all records for a given set of customers, as to mark the records as having been billed.

4.2 CDD Components

Distribution Manager The RAE server is responsible for placing a customer-id upon each processed call detail record. The mapping table is used to map each processed call detail record onto the database server where the record should be stored. Thus, all incoming processed call detail records need to go through a distribution manager that looks up customer-id field of the call record, looks up the mapping table, and then sends the processed call detail record to the database server responsible to store processed call detail for that customer-id.

Query and Update Manager A separate query and update manager is responsible for processing queries and updates to processed call detail data. Bill rendering is done by client billing applications that read processed call detail data from the CDD, and update the CDD to indicate that a set of call records have been billed. These queries and updates apply to a set of customers at a time, and go to the CDD through the query and update manager interface. The mapping table is used by the query and update manager to direct the queries and updates to the appropriate database server. Given a query for a set of customer-ids, the query manager must decide which set of database servers hold the data for the customers, direct the queries to these servers, and collect the answers from each server before presenting these to the querying client.

Database Server Each database server stores some processed call detail data on disk, and some on a tertiary storage device, such as an optical juke box. For instance, 3 months of data might be stored on disk for quick random access, and the previous 21 months of data might be on an optical jukebox, for a total of two years of on-line data.

Scalability The CDD architecture is scalable with respect to increasing number of customers, increasing call volume, and increasing peak rates at which processed call detail records come into the CDD.

For example, assuming that the current database servers are working to capacity, more customers can be added by adding another database server, and assigning the new customers to the new database server. The mapping tables are then modified to reflect the assignment of the new customers. The architecture does not limit the number of servers that can be added into the system, since there is no requirement for the servers to communicate with each other.

As calling volumes increase, and the ability of a database server to service its current assignment of customers goes down, we can scale the system by re-assigning some existing customers to a new server. Similarly, if some database servers are very heavily loaded, and others are lightly loaded, we can do load balancing by re-assigning customers from the heavily loaded servers to the lightly loaded servers.

Multiple copies of the distribution manager can also work in parallel, as illustrated in Figure 2. Each copy of the query manager has its own copy of the mapping table, and runs an identical piece of software. The incoming processed call detail records can go to any copy of the distribution manager, and then be sent to the appropriate database server. The query and update managers are also similarly replicated.

4.3 A Case Study

A call detail database system based upon the architecture above was prototyped, and then benchmarked using well known hardware and relational database software systems [NOLC95]. Telephone call records representing a few months of calls made by tens of millions of customers were targeted, which meant that hundreds of millions of calls would come into the system every month. Further, several hundreds of millions of calls would also be updated every month. Queries came from billing applications and customer care agents. The call detail records had to be indexed by a few fields to answer these queries.

The benchmark [NOLC95] tested the performance of insertions, deletions, updates, queries, backup, and recovery. The benchmark showed that our architecture would easily meet the performance needs of the call detail system using just a few medium-sized servers from a leading hardware manufacturer.

5 Integrated Customer Profile

The *integrated customer profile* (ICP) is a large database that contains information about subscribed services as well as summary data. A copy of the part of the ICP profile and summary data that is essential to do real-time pricing is also kept in memory in the RAE server. In this section, we will only discuss the main ICP store.

The ICP contains one logical record for every accounting unit, commonly known as a customer. An accounting unit (customer) can be a telephone number, an individual, a business entity that includes several sub-accounts, or a journaling unit. In other words, there is an ICP record for every accounting unit to which calls are charged or accounted for. In this section, we will use customer in a generic sense to refer to an accounting entity.

Given the generic nature of a customer, a customer "type" field is used to distinguish between different types of accounting entities represented by the customer. As an example, the type field can have values *residential*, *small business*, *medium business*, *large business*, *journaling*, *marketing panel*. The type *journaling* means that the accounting entity is useful only for journalization, that is, distributing profits and discounts amongst various internal organizations. The type *marketing panel* means that the accounting entity is being used for a marketing study. A call can be linked to several accounting entities, and be processed based upon the types of these entities. The ICP also supports introduction of new types.

The profile record for a customer includes a customer-id, and identifiers for each service the customer has subscribed for. For example, a long distance service may use the telephone number as an identifier. The cellular service may use the cellular number as an identifier. A frequent caller point service may also use the telephone number as an identifier. It is important that one be able to look up a customer's record given any of these identifiers. The ICP builds indices on all the identifier fields. In addition to the identifiers, the ICP includes, for each customer, all other data that the services chose to collect and keep about the customer. Typically the data includes the customer's name and address, the discount plans the customer is enrolled in, and several summaries on the usage and charges incurred by the customer.

5.1 Deriving the Integrated Customer Profile

We assume that each company or organization offering a service maintains its own *service-specific customer profile* (SSCP). For instance, the providers of long distance service, local service, and cellular service may all manage their own service specific customer profile databases (Figure 3). The SUNRISE II ICP architecture allows each service provider to retain control over its own data, while allowing them to bring their data together for advanced billing and information management services.

Our idea is to integrate the customer data from the SSCPs into an integrated customer profile (ICP) using materialized views. The ICP is defined as a view over a normalized form of the SSCPs. The view is then materialized in the ICP store, as shown in Figure 3. Three SSCPs are shown in

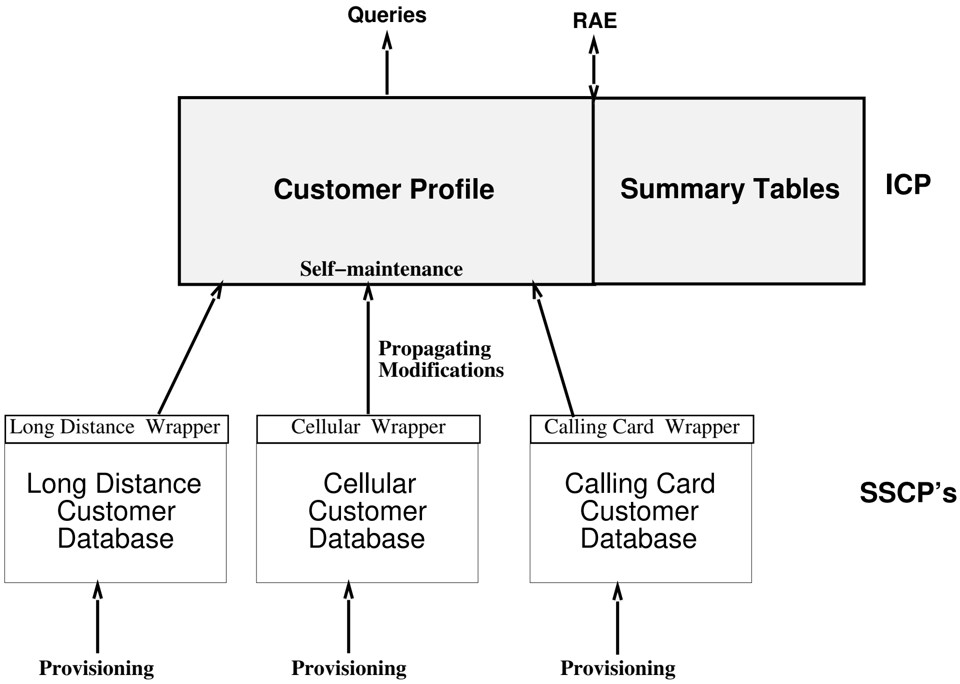

Figure 3: Integrated Customer Profile.

Figure 3, for long distance, cellular, and calling card services. Materialization means that a copy of the data is brought in from the SSCPs and stored locally in the ICP store. Materialization is important to achieve good performance for queries over the ICP. Using the materialized view, queries can be answered locally, without reference to the underlying SSCPs. Another benefit of using materialized views is that it becomes easy to maintain the ICP in response to changes occurring at the SSCPs.

The ICP view is nominally defined as a join or outer-join view of the customer tables along customer-ids. A high level language, such as SQL [ISO93], can be used to define the view. One problem that arises is that the different SSCPs may not use the same customer-id's (due to historical or independence assumptions), so that there is no common join field between the SSCPs. However, joins can be made using other fields, such as name, address, and social security numbers, or by populating a separate table at the ICP that relates the customer-id with identifiers used by the SSCPs. Another problem is that the different SSCPs may use different schemas, making it difficult to combine the information using a view. Wrapper modules [HGMW$^+$95, Wid95] are used around each SSCP to resolve the schema differences, and to map various fields into the form needed in the ICP. The ICP view is then defined over the schema exported by the wrappers.

Maintenance of the ICP: The individual service providers have control over the SSCPs. Consequently, provisioning of customers is done by the individual service providers, and leads to modifications of the relevant SSCPs. Updates to SSCPs must be applied to the ICP so that the information used for billing is accurate, and correctly reflects all the discount plans and customers being billed.

Having the ICP defined as a materialized view helps in automating the process of maintaining the ICP. We require that each SSCP capture the changes applied to it, and regularly transmit a log of all modifications to the ICP. The ICP then uses incremental view maintenance techniques [GM95, GJM96, HZ96] to apply the changes to the ICP. Given the distributed nature of the SSCPs and ICP, it is valuable to make the ICP a self-maintainable view to make view maintenance efficient [QGMW96, GJM97], as will be discussed briefly in Section 5.4.

Given the above architecture, a certain time can elapse before changes made to an SSCP are applied to the ICP. The amount of time needed depends upon the ability of the SSCP to provide a log of its modifications to the ICP. For example, if an SSCP provides the log daily, changes requested by customer during the day cannot be effected until the next day. If such a delay is not acceptable, then the SSCP must implement a mechanism to propagate changes more often.

It is also possible to do some types of updates directly to the ICP, and then have these updates be translated into updates to the individual SSCPs. For example, an integrated customer care system [GMM+95] may provide customer care agents with access to the ICP, and allow them to update the ICP.

5.2 Summary Data

The summary data in the ICP keeps track of various subtotals of charges and usage made by customers over a period of time. Summaries are computed by the RAE server, and are used to support the real-time pricing activity of the RAE server. The summary data is downloaded into the ICP from the RAE at regular intervals, usually every few hours or a day.

Examples of summary data are number of call minutes since the last bill, total cost of calls made on any particular day, the charges accumulated on international calls, and the number of frequent caller points. The chronicle data model (Section 3.4) ensures that the amount of summary data is independent of the number of calls made or billed to a customer.

The ICP can also store summary data for the last several months, while the RAE only stores the summaries needed to do pricing for the current month in memory. The RAE updates the summary data after each telephone call, but these changes are not reflected in the disk based ICP until a snapshot is taken.

5.3 Fragmentation of the ICP

The ICP is a huge data store, and can contain several terabytes of information. The ICP is stored in a horizontally fragmented manner, mirroring the fragmentation of processing between the RAE servers, and the fragmentation of call detail data. A mapping table is used to direct updates and queries into the appropriate ICP server, as was done for the call-detail database.

5.4 Self-Maintainable Views

We consider the scenario of Section 5.1, where a materialized view defined over remote databases needs to be maintained, Often, it is expensive or impractical to obtain much more than just a periodic report of local updates from each underlying database. Even if an underlying database does not provide relational query support, one can require it to provide a log of changes it makes. Under such circumstances, it becomes crucial that the integrated view be *self-maintainable*, meaning that view maintenance should be possible without requiring access to any underlying database, and without access to any information beyond the view itself and the log of the changes. The discussion below is derived from [GJM96, GJM97, QGMW96].

We define the self-maintenance problem as the problem of maintaining a view in response to insertions, deletions, or updates (collectively referred to as modifications) using only the view and the set of changes to the referenced relations (without accessing the complete relations).

Definition 5.1 ((Self Maintainability with respect to a Modification)): A view V is said to be self-maintainable with respect to a modification type (insertion, deletion, or update) if for all database states, the view can be maintained in a manner that does not require access to the base relations, in response to a modification of that type to the base relations. ■

Whether a view is self-maintainable depends on both the definition of the view and on the type of modification, and also on other finer distinctions such as which attribute of a relation is updated, or the actual value of the modified attribute, or the presence of functional dependencies and other integrity constraints.

Insertions: All SP (select-project) views are self-maintainable. SPJ (select-project-join) views are self-maintainable for insertions only under very limited circumstances. In fact it is not possible to self-maintain an SPJ (select-project-join) view joining at least two distinct relations upon an insertion into a component relation. However, full outer-join views are usually self-maintainable with respect to insertions.

Deletions: An SP or SPJ view is self-maintainable with respect to deletions if the base relations have a key, and the keys are retained in the view. Thus, if the ICP is defined as an SPJ view, and each SSCP has a key, and the component keys are retained in the ICP view, the ICP can be self-maintained in response to deletions to SSCPs.

Updates: An SP or SPJ view is self-maintainable with respect to updates if either (1) the updated attributes are irrelevant to the view definition, or (2) the updated attributes are not used in a selection condition, the base relations have a key, and the keys are retained in the view. Thus, if the ICP is defined as an SPJ view, and each SSCP has a key, and the component keys are retained in the ICP view, the ICP can be self-maintained in response to updates to attributes not involved in selection conditions.

Making views self-maintainable: Given the performance benefits of self-maintainability, it is important to have all views in the ICV be self-maintainable. When a desired view is not self-maintainable, it is possible to define extra views so that the set of views, taken together, is self-maintainable. For example, a simple solution to make an SPJ view self-maintainable is to define appropriate SP views over base relations and materialize these in the ICP store [HZ96]. Another solution is to use referential integrity constraints to define a set of semi-join views to be materialized at the ICP store [QGMW96]. Outer-join views can be made self-maintainable simply by retaining all key attributes and attributes used in conditions in the view.

6 Service Creation Platform

An important feature of SUNRISE II is that new features (*e.g.*, discount plans, promotions, marketing information needs, journaling information requirements, or any new type of summary computation) can be introduced and implemented quickly, in a matter of hours or days. Similarly, existing features can be modified quickly. It is also feasible to add a new billing stream into the system, and utilize SUNRISE II as a convergent biller. Further, the architecture enables the new features to be added into the system without stopping the system.

6.1 Architectural Support for New Feature Creation

At the architecture level, use of table-driven features, and a flexible customer profile structure, provide the critical support for adding new features.

Features are specified using tables, and these tables are used to drive the rating, discounting, and summary computation modules in SUNRISE II. For example, a discounting feature is represented by tuples in a discounting table that specify the parameters guiding the discounting process. A separate function in the system is responsible for interpreting the discounting table. A new discount plan that fits into the interpretation of a pre-existing discounting table can be implemented by simply making entries into the table. However, if the new discount plan does not fit into the interpretation of a pre-existing discounting table, a new table must be defined during service creation. The service creation platform provides support for creating new tables, and for defining new function to interpret such newly defined tables.

The in-memory copy of the integrated customer profile is stored fragmented, so that storage is used up only for the features subscribed by the specific customer. For each customer, a list of pointers to each relevant data field for the customer is stored separately. New fields for a customer can now be added simply into a vertical fragment, along with insertion of pointers into the list maintained for the customer. A consequence of the organization above is that when defining a new feature, we only need to add some fields into an existing table, or to define a new table. The data in the tables needs to be changed only when an individual customer subscribes to the new feature.

6.2 Convergent Billing

Convergent billing refers to taking as input several billing streams from disparate services, and producing a single bill as if all the input events belonged to one billing stream. For instance, consider the scenario where separate companies offering limousine service, car rental service, and cellular phone service get together and decide to offer customers an integrated service consisting of all three of the above, and is going to bill customers at the end of the month for combined usage of all three services. The conglomerate might decide to offer discounts on limousine rides to an airport if the customer has a car rentals associated with the same business trip, and uses the cellular phone to call home at least five times during the trip. The billing system to support such convergent billing would take as input three different types of transactions, and be able to price and cross-discount between them. Further, flexibility demands that the billing system be able to incorporate a fourth service, such as camera rentals, that may be added later.

SUNRISE II can support adding additional services into a convergent billing system, even if the additional services are provided by third party service providers. To add an external stream, a *call* record of the appropriate type must be defined, followed by any rating, discounting and summary computations needed for pricing and cross-discounting the new service.

6.3 Creating New Features

The service creation platform provides a graphical interface to define new features. Each new feature must specify the following functions, either through tables, or directly as functions to be dynamically linked into the system:

- A subscription function, to be executed when a customer subscribes to the feature.
- An unsubscription function, to be executed when a customer unsubscribes to the feature.
- Definitions for zero or more fields, including summary fields, required in the ICP by the feature.
- Definitions for zero or more fields required in the processed call detail record on account of the feature.
- An end-of-billing-period function, to be executed at the time of generation of the customers bill.
- Queries over summary and/or detail data provided by the feature.

6.4 Feature Interaction

To decrease software development (and maintenance) cost, and to enhance flexibility, there is a move to make discounting features "rule-based" wherever possible. Rules are a mechanism to declaratively specify the implementation of each feature, independent of other features that might exist in the system. As an attempt is made to capture in rules the involved semantics of features, rule conflicts and ambiguities are inadvertently created.

For example, consider a scenario where frequent calling points (`points`) are awarded to a customer on each call based upon the charge for the call. Further, the company offers double `points` to their gold customers. The wireless service now might offer a promotion that provides double the regular `points` for all wireless calls.

The features now conflict! Should wireless calls made by gold customers get double the regular `points`, triple the regular `points`, or four times the regular `points`? The right answer depends upon the marketing department, and it is their job to specify how such a feature conflict should be resolved. However, the problem is that the marketing department may not be aware of this conflict, or of others that might exist even after the current conflict is resolved.

We have developed [JMM96] a framework to specify, using meta-rules, how the conflicts between features should be resolved. The framework models each feature through one or more rules. Given a set of rules and meta-rules, we can then determine whether all conflicts between features have been resolved by the meta-rules. If not, the marketing department has to specify extra meta-rules, and perhaps change the rules, so that conflicts do not occur. Our meta-rule framework depends on an external data flow analysis module to detect possible conflicts between the rules. A summary of the framework is given below. The reader is referred to [JMM96] for details.

6.5 Meta-Rule Example

EXAMPLE 6.1 Consider a telecommunications company offering frequent calling points to customers. For purpose of this example, suppose that the customer and call information is stored in an object-oriented database with the following two classes:

1. A `Customer` class with several attributes including `total_points`, representing the total frequent caller points accumulated by the customer, and `status`, representing the current membership grade for the customer, such as Basic, or Gold.

2. A `CallDetail` class with one instance per call per customer, with attributes `cust_id`, `origin`, `destination`, `date`, `type_of_service`, `price`, `points`, among others. The `price` attribute gives the price charged to the customer for this call. The `points` attribute records the number of frequent caller points earned by the customer on account of this call. The `cust_id` attribute is a pointer to an instance of the `Customer` class.

One frequently applied update to this database is the insertion of a new instance of the `CallDetail` class, recording the fact that a customer made a call. For simplicity, we assume that the price charged for a call is computed at the time of insertion of the call, and is available in the `price` attribute immediately. Associated with each `CallDetail` object, and fired upon creation of the `CallDetail` object, are triggers that cause an appropriate value of the `points` attribute to be computed, and cause the `total_points` attribute of the associated `Customer` object to be updated. Some of the triggers written for this purpose are described below.

(*a*): `UPON CallDetail`
    if (`TRUE`)
    `points` = `price`; .

Rule [*a*] does the basic point computation based upon the price for the call.

(*b*): `UPON CallDetail`
    if (`points` < 2)
    `points` = 2 ; .

Rule [*b*] ensures that at least 2 points are awarded for each call made.

(*c*): `UPON CallDetail`
    if (`type_of_service` == "wireless")
    `points` = 2 * `points` ; .

Rule [*c*] awards double points for wireless calls.

(*d*): `UPON CallDetail`
    if (`cust_id`→`status` == "gold")
    `points` = 2 * `points` ; .

Rule [*d*] awards double points to "gold" customers.

(*e*): `UPON CallDetail`
    if (`cust_id`→`total_points` >= 1,000)
    `points` = 2 + `points` ; .

Rule [*e*] awards an extra 2 points to any customer with more than 1,000 points in their account already.

(*f*): `UPON CallDetail`
    if (`destination`.areaCode = "415")
    `points` = 2 + `points` ; .

Rule [*f*] awards an extra 2 points for calls into the "415" area code as a special promotion.

```
(g):  UPON CallDetail
          if (TRUE)
          cust_id→total_points =
                  cust_id→total_points + points ; .
```
Rule $[g]$ accumulates into the customer account the total points awarded.

■

It is easy to see that these rules have interactions. For instance, if the rule [g] were executed before all the other applicable ones, then the result stored for total points in the CUSTOMER class would be wrong. Similarly, [a] must have completed execution before any of the other rules can execute properly.

Some controls on rule execution, such as those discussed in the preceding paragraph, are required simply to produce a meaningful result. Other controls may be applied to select between different possibilities. For instance, applying [f] before [d], will give gold customers 4 points extra, rather than 2 points extra, if "415" is their terminating area code.

Often, gold customers may have more than 1,000 points in their account. We may desire that these customers get only the double points and not the 2 point bonus in addition. In other words, rule [d] and rule [e] disable each other, with respect to a specific customer. Further, for a gold member with more than 1,000 miles we want to prefer the gold bonus to the 1,000 mile bonus.

Our approach to the management of these interactions between rules is through the declarative specification of meta-rules. For instance, a meta-rule could state, in the example above, that only one of [d] and [e] should execute when both are applicable to a customer. Our problem then is how to specify and implement meta-rules that select a subset of applicable rules for execution in any context, and further determine in which order the rules in the subset should be executed.

6.6 Rules and Meta-Rules

An active rule system $\mathcal{S} = \langle V, \mathcal{M} \rangle$ consists of a set of rules, V, and a set of meta-rules, $\mathcal{M}$. We first present our model for active rules, and then introduce a language for specifying meta-rules to control interaction between rules.

6.6.1 Rules

A rule comprises an event, a condition, and an action, and is associated with a particular object in the database. Each rule is *activated* upon an event occurrence, such as creation of the object. The set of rules associated with an object that can be activated upon a common event is denoted as V.

When a rule is activated, its condition is checked. If the condition of a rule evaluates to true, the rule is said to be *fireable*. A fireable rule may or may not be *executed*, depending upon the meta-rules specified to control execution of the rules.

The set of fireable rules associated with a particular object (or tuple) is denoted as I, or the input set (input for the rule selection algorithm). The subset of I that actually executes is denoted as O, or the output set (output from the rule selection algorithm). Clearly, $O \subseteq I \subseteq V$. An ordering for all the rules in O is determined, and all these rules are executed in the specified sequence. This execution model is similar to that of [DBB+88, GJ91].

EXAMPLE 6.2 Consider the active rules $[a] \cdots [g]$ in the frequent caller point system of Example 6.1. All these rules are activated upon the creation of a CallDetail object. Thus $V = \{a, b, c, d, e, f, g\}$.

Rules $[a]$ and $[b]$ are always fireable, since `points` $= 0$ is assumed to be true initially. (Rule $[b]$ may not fire after rule [a] executes, but we do not know this in advance.) Rule $[c]$ is fireable if the call is a wireless call. Rule $[d]$ is fireable if the call is made by a "gold" customer. Rule $[e]$ is fireable if the call is made by a customer with more than $1,000$ total points. Rule $[f]$ is fireable if the call terminates in the "415" are code. Rule $[g]$ is always fireable. Thus, the input set I depends upon the newly created flight detail object. ■

Because we analyze rules after the fireable set has been determined, from now on we treat rules as atomic objects with no internal structure. We do not try, for example, to detect mutually exclusive conditions or rules that will never become fireable.

6.6.2 Meta-Rules

Let $\mathcal{M}$ be a set of *meta-rules* that control the interaction and execution of the rules in V. These meta-rules are used to (1) select the set O from I, and (2) to order the resulting execution set. The rules in the set $I - O$ are not executed, as a result of the restrictions imposed by the meta-rules. Meta-rules are of four types, as listed below, where, $A, B, C, D, \ldots \in V$:

Positive requirement meta-rules : $A \supset B$; If rule A executes, then rule B must execute as well.
In the frequent caller example, rule [c] requires that rule [a] must also execute. To express this constraint, we write a meta-rule $[c] \supset [a]$.

Disabling rules : $\overline{AB}$; Rules A and B disable each other; only one of them may execute. The meta-rule $\overline{A}$ is a special unary case of this rule and means that the rule A can never be executed. It can be viewed as short hand for $\overline{AA}$.

In the frequent caller example, only one of rules [d] and [e] should be allowed to execute.

Preference rules : $B > A$; If A and B are both fireable, then B should not be dropped from the execution set unless A also is. In other words, if A and B are both fireable, and there is some reason why A and B cannot both be executed, then we prefer to execute B rather than A.

In the frequent caller example, when rules [d] and [e] are both fireable, execution of both together is disallowed by the disabling meta-rule $\overline{[d][e]}$. At this point, we prefer to execute rule $[d]$ rather than rule $[e]$, that is, $[d] > [e]$.

Scheduling rules : $A \prec B$; If A and B both are selected for execution, execute A before B.

In the frequent caller example, when rules $[a]$ and $[b]$ are both executable, we can require the rule $[a]$ execute before $[b]$ by stating $[a] \prec [b]$.

To understand the difference between positive requirement and preference rules, consider the frequent caller example.

EXAMPLE 6.3 As mentioned earlier, $[d] > [e]$ means that we must choose rule $[d]$ when both rules $[d]$ and $[e]$ are fireable but we have to choose only one. If the fireable set contains only $[e]$, then we want to execute $[e]$. If the fireable set contains only $[d]$, then we want to execute $[d]$. On the other hand, $[c] \supset [a]$ means that for rule $[c]$ to execute, rule $[a]$ must execute. If only rule $[c]$ is fireable, then $[c]$ cannot execute. However, if only rule $[a]$ is fireable, then $[a]$ can execute, and if both rules $[a]$ and $[c]$ are fireable, but we must choose one of these, then rule $[a]$ can execute. ■

EXAMPLE 6.4 For the frequent caller example, the following meta-rules capture the desired policy to resolve the interactions stated in Section 6.5.
$b \supset a$: Rule [b] positively requires rule [a].
$c \supset b$, $d \supset b$, $e \supset b$, $f \supset b$, and $g \supset b$: Rules [c], [d], [e], [f], and [g] all positively require rule [b].
$\overline{de}$: Rules [d] and [e] cannot execute together.

$d > e$: If both [d] and [e] are fireable, then prefer [d].

$a \prec b$, $b \prec c$, $b \prec d$, $b \prec e$, $b \prec f$, $b \prec g$, $c \prec f$, $c \prec g$, $d \prec f$, $d \prec g$, $e \prec f$, $e \prec g$, and $f \prec g$: Whenever a listed pair is executed, one must be scheduled before the other. ■

The formal semantics of a set $\mathcal{M}$ of meta-rules is defined by considering the models for the set $\mathcal{M}$.

Definition 6.1 (Model): A *model* for a set $\mathcal{M}$ of meta-rules is a quadruple $\langle V, I, O, \omega \rangle$ where V is a set of rules (treated as atomic objects for our purposes), $O \subseteq I \subseteq V$, and ω is a total order on the set O.

We say that $\langle$V,I,O,$\omega\rangle$ *satisfies* meta-rule μ if:

1. μ is of the form $A \supset B$ and if $A \in O$, then $B \in O$.
2. μ is of the form $\overline{AB}$ and $\neg(A \in O \ \& \ B \in O)$.
3. μ is of the form $B > A$ and if $AB \subseteq I \ \& \ A \in O$, then $B \in O$.
4. μ is of the form $A \prec B$ and if A and B both appear in O, then A appears before B in the total order ω.

The quadruple $\langle V, I, O, \omega \rangle$ is a model for the set $\mathcal{M}$ of meta-rules if $\langle V, I, O, \omega \rangle$ satisfies every meta-rule $\mu \in \mathcal{M}$. ■

EXAMPLE 6.5 For the frequent caller example, one may verify that the quadruple

$$\langle \{a,b,c,d,e,f,g\}, \{a,b,c,d,e,g\}, \{a,b,c,d,g\}, \\ (a \rightarrow b \rightarrow c \rightarrow d \rightarrow g) \rangle.$$

is a model. The term $(a \rightarrow b \rightarrow c \rightarrow d \rightarrow g)$ specifies the total order ω between the rules in the output set. ■

6.6.3 Analysis of a Meta-Rule System

Given an active rule system $\mathcal{S} = \langle V, \mathcal{M} \rangle$, [JMM96] gives algorithms to determine the following:

- Given a rule $A \in V$, is it possible that A does not execute for any input set? What are all such *dead* rules?[1]

[1] A rule may never fire because it has a condition predicate that can never be true, or is triggered by an event that can never occur. Such analysis, of the semantics of the rules themselves, is outside the scope of this paper. When we say *all* dead rules, we mean all rules that are dead on account of the specified set of meta-rules.

- Given a pair of rules $A, B \in V$, can A and B execute together for some input? What are all such A, B pairs?[2]
- Is a non-empty solution O possible for at least one valid non-empty input set I?
- Is the set of meta-rules guaranteed to result in a unique total order ω for every output set O for every fireable set I?
- Are some of the meta-rules in $\mathcal{M}$ redundant?

7 Conclusions

We have presented an innovative architecture to do billing and information management for telecommunications services. The architecture enables convergent billing, real-time pricing, quick creation of new features and services, innovative billing options, and advanced fraud control.

The proposed SUNRISE II architecture consists of four major components — Real-time pricing, call detail database, integrated customer profile, and a service creation platform. We described each of these components, and explained the technology underlying the components. References to detailed discussion of the technical developments were given.

We hope that the SUNRISE II architecture will influence the development of the next generation of billing and information management systems in the telecommunications industry.

References

[DBB+88] Umeshwar Dayal, Barbara Blaustein, Alex Buchmann, U. Chakravarthy, Meichun Hsu, Rivka Ladin, Dennis R. McCarthy, Arnon Rosenthal, S. Sarin, Michael J. Carey, Miron Livny, and R. Jauhari. The hipac project: Combining active databases and timing constraints. *ACM-SIGMOD Record*, 17(1):51–70, March 1988.

[DEB95] *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2), June 1995.

[GJ91] Narain Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In Guy M. Lohman, Amilcar Sernadas, and Rafael Camps, editors, *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 327–336, Barcelona, Spain, September 3-6 1991.

[GJM96] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In *Proceedings of the Fifth International Conference on Extending Database Technology*, Avignon, France, March 1996.

[GJM97] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Maintenance and self-maintenance of outer-join views. Unpublished manuscript, February 1997.

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing* [DEB95], pages 3–19.

[2] As in the case of dead rules, we do not analyze whether there actually is a database state and event occurrence for which the condition predicates of the two rules can actually be satisfied simultaneously. We merely capture whether such simultaneous satisfaction is possible given the set of meta-rules. In fact, if the application semantics indicate that two rules can never fire together, it is useful to capture this as a meta-rule.

[GMM+95] Narain Gehani, William McKenna, Inderpal Singh Mumick, Michael RabinovichLatha Colby, and Akira Kawaguchi and. Welcome – an architecture for integrated customer care systems. Technical report, AT&T, 1995.

[HGMW+95] Joachim Hammer, Hector Garcia-Molina, Jennifer Widom, Wilburt Labio, and Yue Zhuge. The Stanford Data Warehousing Project. In *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing* [DEB95], pages 41–48.

[HZ96] Richard Hull and Gang Zhou. A framework for supporting data integration using the materialized and virtual approaches. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of ACM SIGMOD 1996 International Conference on Management of Data*, Montreal, Canada, June 1996.

[ISO93] ISO_ANSI. ISO-ANSI working draft: Database language SQL3, 1993.

[JKMS94] H. V. Jagadish, Rama Kanneganti, Inderpal Singh Mumick, and Avi Silberschatz. SUNRISE: An architecture for billing, rating and information services in a telephone network. Technical Report 94-0401-29-TM, AT&T Bell Laboratories, April 1994.

[JMM96] H. V. Jagadish, Alberto O. Mendelzon, and Inderpal Singh Mumick. Managing rule conflicts in an active database. In *Proceedings of the Fifteenth Symposium on Principles of Database Systems (PODS)*, Montreal, Canada, June 1996.

[JMS95] H. V. Jagadish, Inderpal Singh Mumick, and Avi Silberschatz. View maintenance issues in the chronicle data model. In *Proceedings of the Fourteenth Symposium on Principles of Database Systems (PODS)*, pages 113–124, San Jose, CA, May 22-24 1995.

[MQM97] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of summary tables in a warehouse. In Patrick Valduriez and Joan Peckham, editors, *Proceedings of ACM SIGMOD 1997 International Conference on Management of Data*, Tuscon, AZ, May 1997.

[Mum95] Inderpal Singh Mumick. The Rejuvenation of Materialized Views. In *Proceedings of the Sixth International Conference on Information Systems and Management of Data (CISMOD)*, Bombay, India, November 15-17 1995.

[NOLC95] NCR, Oracle, AT&T Bell Labs, and AT&T CCS. CCS SUDS Benchmark. Technical report, NCR White Paper, 1995.

[QGMW96] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making views self-maintainable for data warehousing. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, FL, December 18-20 1996.

[Wid95] Jennifer Widom. Research problems in data warehousing. In *Proceedings of the Fourth International Conference on Information and Knowledge Management*, pages 25–30, Baltimore, Maryland, November 1995.