Transactional Intent

Shel Finkelstein, Thomas Heinzel, Rainer Brendle, Ike Nassi and Heinz Roggenkemper
SAP Research
3410 Hillview Avenue
Palo Alto, CA 94304
{firstname.lastname}@sap.com

ABSTRACT

Data state in a data management system such as a database is the result of the transactions performed on that data management system. Approaches such as single-message transactions and field calls [Gray1993] come closer than before/after values to expressing the intent of a transaction, the semantic transformation that should be performed on the data state even if that state is different than what was previously read. But intent is an even higher-level semantic description. This paper illustrates the use of intent-based transactions and processes in several applications, and describes the benefits from exploiting transactional intent. We provide an application framework for intent, and discuss some advanced aspects of intent, including its relationship to apology-oriented computing [Helland2007].

Categories and Subject Descriptors

H.2.4 **[Systems]**: Concurrency, Distributed databases, Query processing, Transaction processing. H.2.8 **[Database applications]**. J.1 **[Administrative data processing]**: Business, Financial, Manufacturing. D.2.11 **[Software Architectures]**: Data abstraction, Patterns. D.1.3 [**Concurrent Programming]**: Distributed programming.

General Terms

Design, Management, Performance.

Keywords

Intent, data management, database, transaction processing, metadata, business applications, business processes, optimization, supply chain management, supply network collaboration, operational business intelligence, apology-oriented computing, events, callbacks, compensation.

5th Biennial Conference on Innovative Data Systems Research (CIDR '11)
January 9-12, 2011, Asilomar, California, USA.

1. INTRODUCTION

A data management system such as a database (DB) contains data, which at any time has a state. As transactions are performed, that state changes. Database state is an interesting materialized view on the database log. Some systems store multiple versions of some data, in which case all the versions are part of the state. Transactions transform the DB state, by doing insert, update and delete operations (or utilities such as loads) on rows (or sets of rows) in tables.

IMS FastPath [Gray1993] provided single message transactions which transformed IMS state by reading and updating data, only holding locks while the transaction was executed. Field call (e.g., increment/decrement) approaches to state transition have this same transformational character, describing an operation (possibly a program with a series of steps) to be performed, rather than a new state. The advantage of operation-oriented transactions (for transactions consisting of a single operation) is that concurrency control to isolate one transaction from another is only required while the operation is executed, avoiding long-running pessimistic locking as well as optimistic concurrency control rollbacks. (Short-term conflicts can be addressed by retrying operations.) Having transactions consisting of single stored procedures is a generalization of this idea.

Operation-oriented approaches describe state transformations, rather than producing a new state assuming (enforced or hoped for) assumptions about old state matching expectations. The *intent* of a transaction can be a series of operations or a description mappable to a series of operations. If the mapping from a transaction's intent to its state transformation is deterministic, then a log of the intents for a sequence of transactions deterministically defines the transformations performed by the sequence of transactions, based on the composition of (transformations defined by) the intents of those transactions.

As a very simple (and frequently cited) example, consider an inventory transaction whose intent is to change the inventory of a bin based on a formula (such as increment/decrement). The semantics of each operation is well-defined, as is the semantics of a sequence of such operations. Normal execution of a sequence of transactions in a given state produces the intended semantics. However, unless the intent of the transactions is recorded, the general semantics of the operations is lost; they were applied in the current state correctly, but could not be applied in changed circumstances. A transformation from old inventory of 10 to new inventory 0 might have subtracted 10, but it also might have set inventory to 0, or doubled it and subtracted 20.

However, intent may be more semantically sophisticated than just an operation (or a series of operations). The business intent might be to fulfill a sales order from a customer (which happens to be for 10 units of a product), possibly taking into account other customer orders and maximizing some overall goal (such as total profit), a higher level of semantics than the subtraction operation; this intent should be recorded, in addition to operations and/or data transitions.

Why record intent if transactions are complete? As we'll see in the application examples discussed in Section 2, things can go wrong and business circumstances change, so compensating transactions/apologies [Helland2007] may be needed, sometimes including re-execution of transactions in a new state. Intent can be stored in the database just like any other data, allowing it to be searched and accessed (subject to authorization).

Section 3 outlines an application framework for handling intent. Section 4 presents an operational business intelligence example, and section 5 discusses some aspects of business process intent. Section 6 briefly discusses some additional implications and considerations for intent motivated by the examples, and section 7 mentions some related work.

2. APPLICATIONS USING INTENT

This section describes some examples of the use of intent in applications.

2.1 Order Entry

When you deal with an on-line merchant and place an order for the items in your shopping cart (such as books), you often see a screen acknowledging that your order has been received, giving you a number identifying the order. You also often receive an email acknowledging that the order has been received. Before your order is taken, there may be preliminary checking to see if your items are in stock, perhaps by having an (not necessarily up-to-date) inventory estimate in individual order entry nodes (which have catalogs and local transaction histories, which are transmitted to fulfillment services for processing).

Although some order entry systems "guarantee" that the item you ordered is available (e.g., purchases of specific seats), many systems merely acknowledge that they recognize your intent, which is to purchase your shopping cart items. There may be reasons why your purchase cannot be honored, such as too many simultaneous orders of an item (if inventory is not strictly managed).

The separation of order entry from order fulfillment probably seems obvious to people, but teaching clients (humans or applications) to accept that separation is a significant step, separating synchronous capture of the purchaser's intent from an asynchronous response from the merchant system describing actions taken to honor that intent. The response may include the dates items will be shipped and indications that some items may be delayed. If a purchaser specifies that certain items should be shipped together, then delays in one item will delay associated items as well. Only by capturing purchaser's explicit/implicit intent (from order, profile, etc.) can the transaction (or set of transactions) be executed correctly. Even if a subsequent apology is needed (e.g., because a shipment was delayed or a warehouse burned), the same intent-based paradigm is applicable.

2.2 Calendar

This is a description of a hypothetical calendar system using intent; perhaps some system with these capabilities exists.

Suppose that I, as manager, want to schedule a Project Review meeting. It must include the project lead and at least 3 out of 4 staff members, and it must be after an architectural review meeting. The value of the meeting is higher the earlier in the week it is scheduled, but it must be scheduled before next Friday.

This is an intent description that captures constraints and informal objective functions ("earlier in the week is better") for my meeting. If the meeting were scheduled for Tuesday at 2pm without capturing the intent, then it wouldn't be possible to automatically reschedule the meeting if the Architectural Review were delayed. Rescheduling might involve moving a lower priority meeting for me or one of the other staff members.

2.3 Supply Chain Production Scheduling

SAP Advanced Planning and Optimizer (APO) [Balla2007] does planning tasks, such as scheduling machine runs (production orders) to produce finished products. Machines have given capacities; jobs require materials (which may have to come from other jobs). Constraints (e.g., latest delivery dates) and objective functions (weighted combinations of time and costs) are specified, and APO optimizes scheduling to maximize (heuristically) the objective function while meeting the constraints.

Requests to APO model intent either to produce product for inventory stock, or to deliver products for customers. APO acknowledges these requests. Production jobs to fulfill the requests may be scheduled incrementally based on requests taking into account existing resource schedules. Periodically, global optimization may be performed across all jobs. Because intent is captured, not just proposed job fulfillment schedules, such incremental and global scheduling is possible across all jobs, adjusting schedules when necessary for new high priority jobs, which may delay previously entered lower priority jobs.

There is also intent (which we call meta-intent in section 6.3) in the way administrators define master data for the scheduling algorithm, so that certain customers, products, locations and organizations have high priorities.

2.4 Supply Chain Business Interactions

SAP's Availability-to-Purchase (ATP) [Balla2007] is part of Supply Chain Management. To initiate a purchase, a purchaser issues a request for merchandise to a supplier, specifying quantities, delivery dates, quality, etc. The supplier responds with a term sheet that may specify multiple alternatives for quantities, dates and prices of items that are available to purchase, as well as term sheet acceptance deadlines. The purchaser may submit a purchase order based on the term sheet, and the supplier can create an internal sales order and schedule deliveries after the purchase order is accepted.

Each of these steps involves an expression of intent between purchaser and supplier, which enables parties to deal with exception cases and issue/handle apology events. For example, while waiting for the actual purchase order after sending a term sheet, the supplier might choose to reserve quantities only for a limited time (or not at all), and the supplier might have to apologize to the purchaser if quantities are not available after the purchase order is submitted. Later in the process, a supply delivery may be delayed due to a manufacturing problem, just as merchandise delivery may be delayed in the order entry example. Term sheets (describing quotations for items available to purchase) may become invalid for business reasons, e.g., because the purchaser is no longer eligible for discounts or because they time out. Capturing intent supports both better optimization across the set of intents for all sales order, not just based on current schedules, as well as compensation actions (which are forward actions, not rollbacks) if constraints for a particular sales order no longer can be met due to higher priority sales orders.

2.5 Supply Network Collaboration

Applications such as SAP's Supply Network Collaboration (SNC) [Hamady2009] handle collaborative negotiations between companies. For direct material replenishment, price is negotiated up-front but terms for delivery times and quantities can frequently change from both demand-side and supply-side. SNC records current values and past histories for interactions between purchasers and suppliers. A supplier may send an offer to a customer, expressing terms to sell; a purchaser may send an offer to a supplier, expressing terms to buy. These are independent, offers. Either party may propose an agreement based on the terms expressed in the other's offer; for example, the purchaser may send an offer to buy on the supplier's terms. The supplier decides whether to confirm the sale on the terms that it offered; business conditions may have changed. If the supplier confirms, then there is an agreement… unless one of the parties subsequently cancels, requiring handling of that cancellation by the other party.

The communicating purchaser/supplier state machines for SNC record and exploit intent for both business parties (supplier and purchaser). For example, if a supplier does not confirm an agreement, or reneges on terms, the purchaser can determine what the intent of the purchase is, and decide whether to pursue new terms with that supplier or another supplier. Purchaser-side event handlers (which are rule-based) determine which deviations in supplier confirmations should automatically be accepted, and which a human being needs to review.

If the purchaser only knew about the planned delivery (data derived from negotiations) or about the logical operation performed (acceptance of supplier's terms), there would not be enough information to handle the cancellation event (apology) properly. Knowing the purchaser's intent to obtain delivery of merchandise by a given date enables the purchaser's system to deal with the cancellation. Mechanisms and data for doing this are discussed in the next section.

3. APPLICATION FRAMEWORK FOR INTENT

Applications and application frameworks exploit the capabilities of the data management layer of a system, but are not themselves part of the data management layer. Delivering transactional intent for the applications described in section 2 requires an application framework supporting intent. This framework can be built on top of existing data management functionality, although optimizing data management for intent may be valuable, particularly for some of the more advanced capabilities described in section 4.

This section gives a very informal description of what intent is and what the requirements are for an application framework that handles intent. There are many alternative ways that intent can be expressed formally, and many frameworks that meet these requirements, and a single business application suite can support multiple alternative implementations.

3.1 Intent expressions

Definition: **Intent** is an expression of the **goals** and **constraints** that should be met by a business transaction or business process, optionally with **optimization parameters** (e.g., objective function parameters or priorities). There also can be **satisfaction events/callbacks** associated with satisfying the intent and failing to satisfy the event, both when the intent is initially satisfied/not satisfied, and when there are changes in how/whether the intent is satisfied.

Intent for a given problem domain implementation may be expressed using a domain specific language. The expression of intent may be imperative (e.g., explicit code to subtract 10 from Inventory as long as result is zero or more), declarative ("schedule a meeting having the following participants, occurring after an architecture review meeting but before Friday") or some combination of declarative and imperative. The only requirement is that intent be "understood" by the application framework intent optimization engine.

3.2 Intent optimization engines

An application framework for executing intent includes an **intent optimization engine** that determines which intents will be satisfied and how such intents will be satisfied (intent execution plans). Intent may be regarded as metadata; it doesn't describe what has happened (data) or what is going to happen (plan or projections); instead, it describes what the **intent submitter** would like to happen. Any intent execution plan that achieves the intent's goal while meeting its constraints satisfies that intent, and should be acceptable to the intent submitter. But the intent optimization engine heuristically optimizes (based on objective functions or priorities) across all intents.

For example, an intent optimization engine for Supply Chain Production Planning would schedule materials and machine runs to produce products (production orders). Optimization parameters such as priorities and objective function parameters are used by the optimization engine to make scheduling decisions. A scheduling engine that chooses intents based on highest priorities would expect priorities as its optimization parameters. Other intent optimization engines might maximize an objective function total across all intents whose constraints are met, perhaps subtracting penalties for intents that cannot be satisfied. For example, Supply Chain Product Planning might maximize total revenue or profit. More complex optimization strategies are also possible, such as maximizing profit while ensuring that all high priority jobs are finished on time. On the other hand, a very simple optimization engine could use a rule-based approach to satisfy constraints without using priorities or objective functions.

Some optimization parameters might be administratively determined, such as the business importance of the user, organization or the process instance submitting the intent. Other optimization parameters, such as deadlines, and the penalties for not meeting an intent's deadline, might be explicit optimization parameters, or could be expressed as metadata.

When a new intent is submitted to an intent optimizer engine, one scheduling approach is to find the best way to satisfy the new intent without disturbing existing plans. Another approach is to satisfy new high-value intents (with high priority or large objective function contributions) with minimal disruption by rescheduling the plans of one or more intents with lower priority. Of course, such incremental heuristic approaches may not find the maximum utility schedule as determined by the objectives function, so periodic global optimization may be appropriate. In Supply Chain Production Scheduling, global optimization can be expensive when there are many (tens of thousands) resources, products and jobs/intents. Dealing with “apologies” due to rescheduling is another expense, even when products are produced sooner than expected. Tradeoffs between incremental and global optimization depend on the individual organizations and applications involved. Another common scheduling approach (which has both advantages and disadvantages) is to partition the problem into smaller sub-problems which are addressed independently, e.g., by first assigning a job to a particular factory and then doing scheduling within that factory.

3.3 Satisfaction events/callbacks

When an intent optimization engine initially determines how an intent will be satisfied (or that it won't be satisfied), it generates a satisfaction event that notifies the intent submitter (and other interested parties or authorized event subscribers) about its decision. For example, when a calendar appointment is scheduled, people invited to the meeting should receive meeting requests. A simplified approach, which we'll emphasize, requires an intent submitter to specify satisfaction callbacks rather than satisfaction events. Note that there can be separate satisfaction and non-satisfaction events (or callbacks) associated with a given intent. If an appointment's meeting time has to be adjusted subsequently due to a scheduling conflict such as a more important meeting, then a new satisfaction event/callback would be generated. If the appointment has to be cancelled, or delayed past its deadline, than a non-satisfaction event or callback is generated. Such events/callbacks are usually not handled within the transaction generating them, but the framework should guarantee that they will be executed once and only once (using well-known retry and idempotence techniques [Gray1993]).

The event associated with rescheduling or cancelling a delivery or a calendar appointment has been called an apology [Helland2007]. Such events may be frequent in flexible planning environments requiring frequent intent re-optimization. When a purchasing process in Supply Network Collaboration receives a cancellation apology, it must determine how to cope with that event.[1] The purchasing process would compensate by marking the previous agreement as cancelled (and perhaps take other actions, such as tracking supplier's reliability), and then the purchasing process would find another way to meet its intent to received the merchandise by the given date, perhaps by ordering from a different supplier.

3.4 Change metadata

To perform compensation, the purchaser process must know **change metadata** associated with intents (or more specifically, with a plan to satisfy intents), and then perform application-specific methods for compensation, which are always forward-going set of actions, not rollbacks. Change metadata includes three elements:

1. Intents and the **intent execution plans** for meeting those intents, which may involve business objects and sub-intents.
2. A **dependency graph** between business objects, a directed graph labeled with callbacks[2] and conditions in which those callbacks should be invoked. When there's an edge between objects A and B, the callbacks are on object B, while the conditions may involve both A and B.
3. **Version history** for objects, indicating not only value changes but also the intents that led to those changes.

Change metadata describes directed cross-relationships among object versions and intents. As we'll see, these relationships may have been created in different transactions, and even by different business processes running different applications.

3.4.1 Change metadata examples

Example: In the calendar application, meeting M2 may have to occur after another specific meeting, M1, so meeting M2 depends on the timing of meeting M1, and there's an edge between M2 and M2. If M1 is moved to a later time, then the callback on M2 associated with the (M1, M2) dependency edge is invoked.[3] The intent which created M2 (identified in M2's version history)

[1] A purchasing process may also receive notice that a supplier has reduced its confirmation level, which is a weak form of cancellation.

[2] The dependency graph can also be viewed as describing subscriptions to object change events, but we'll mainly use the callback approach in this section.

[3] A more sophisticated implementation might invoke that callback only if M1 now occurs after M2; a less sophisticated implementation might invoke that callback if M1 changes in any way, not just its timing.

might have specified a deadline. If the change in M1 means that M2 can no longer be scheduled before its deadline, then a non-satisfaction callback for M2 will be invoked. The person who scheduled the meeting (or a human or software agent for that person) will be notified and make take further actions, such as moving a higher priority meeting so that M2 can occur on time.

Example: A sales order may have been created in an Availability-to-Purchase application. In Supply Chain Production Scheduling, the dependency graph may include an edge going from that sales order to a machine production order that was created to help fulfill that sales order. This edge indicates that the production order depends on the sales order. If the sales order is cancelled, then the machine production order should also be cancelled, or perhaps redirected to fulfill another sales order, which results in a different dependency graph edge.

But the sales order also depends on the machine production order, so in this case there are edges in both directions.[4] If the machine order cannot be completed because a production line machine failed, then the intent execution plan for the sales order needs to be revisited. The execution plan for the sales order may involve multiple production orders on different machines, each with a sub-intent created to fulfill the overall intent of fulfilling the sales order. The sub-intent behind the failed machine production order could be fulfilled using other production line machines, as long as scheduling dependencies among jobs (which create outputs used by other jobs) are met.

3.4.2 Storing /maintaining change metadata

As the production scheduling example above shows, objects created by one business application may depend on objects created not just by other transactions and business processes but even by other business applications. Hence the change metadata tracked, the means of tracking it, and the methods for handling callbacks require that the set of applications cooperate in a common framework. Let's discuss storing and maintenance of the three elements of change metadata.

3.4.2.1 Storing/maintaining intent execution plans

Storing intent execution plans is relatively straightforward. When an intent execution engine creates a plan satisfying the intent, it stores the intent and the execution plan associated with it. A plan may be hierarchical, involving satisfying sub-intents (such as individual product orders used to satisfy a sales order), each of which has an execution plan. When a plan is updated due to a satisfaction event/callback, the new plan is stored, perhaps keeping the old version (and the reason it was updated) for auditing, historical data mining and predictive analytics.

3.4.2.2 Storing/maintaining dependency graphs

The dependency graph is more difficult to maintain because it is a cooperative data structure built across many instances of different business processes and applications. In a sense, the dependency graph helps unify different applications. An application may access/update many objects; the application writer (or maintainer) must understand which inter-object dependencies matter, as well as how they should be addressed. This knowledge is specific to each application domain, but fortunately that knowledge is often separable by application.

A developer who is writing (or changing) application X should understand which business objects X depends on, as well as how other objects in X depend on those objects. Objects in X may depend on other objects in X (e.g., a production order dependent on other production orders) or on objects that are created or modified outside application X (e.g., a production order dependent on a sales order). Although a developer writing or modifying X[5] may need to know how the objects that X depends on could change (so that X may react to those changes),[6] the developer need not know the internals of the applications that change the objects X depends on. However, the intent of the changes made by those other applications may be worth knowing (and is available in version history, describe in the next subsection).

Writing or maintaining an application requires creating and maintaining the dependency graph, including the callbacks (and code for handling the callbacks) associated with it. Although there could be tools that help with dependency graphs and application separability helps simplify the problem, the creation and maintenance of dependency graphs requires application domain knowledge and some stylistic conventions. For example, if a production order depends on a particular sales order, then including the sales order as a parameter for functions creating or modifying the production order helps the developer describe the dependency.

3.4.2.3 Storing/maintaining version history

Version history, like intent execution plans, is relatively straightforward. The version history for business objects is similar to versioned data in databases, although it has additional information supplied by the application framework. That additional information could identify the intent of the transaction that created each data version, and provenance-like descriptions of subparts of transactions (like production orders created to fulfill a sales order), reflecting their sub-intents. If intent is stored for each transaction, then storing the transaction id for each version

[4] Although there are edges in both directions, a change in one object won't lead to an infinite loop because changes damp out. Infinite loops should be highly unlikely in correct programs because of application semantics.

[5] Writers of specific components of application X should only need to know about object dependencies involving objects in their components.

[6] One coarse way a developer can deal with updates of objects you depend on is to treat each of them as a deletion followed by an insert, which simplifies the set of changes the developer must handle. In practice, that would be an awkward and inefficient approach, hiding the semantics of the update and potentially obfuscating its intent.

identifies the intent of the transaction, but doesn't identify higher level business process intent (discussed in section 5) or lower level sub-intents, such as creating production orders.

Additional "provenance" that could be stored in version history is the events/callbacks processed due to dependencies, including the object that changed, the dependent objects and actions taken by callbacks (which typically involve one or more transactions and intents). As with intent execution plans, keeping versions of this data may be valuable for auditing, historical data mining and predictive analytics.

3.5 Some application framework implementation considerations

In this section we informally described an application framework for intent, including intent expressions, intent optimization engine, satisfaction events/callbacks and change metadata. Change metadata is the most complex part of this, particularly dependency graphs.

We emphasize that the capabilities described in this section are all at the application layer (application framework plus application programming), although the application layer leverages data management capabilities to retrieve and update application data and metadata. Of course, transactional techniques should be used to ensure atomicity of data/metadata updates, as well as the asynchronous events they generate (which are handled outside the transaction). SAP traditionally has handled many aspects of data management (buffering, locking, update queues) in an application layer outside of the database [Finkelstein2008], only touching the database (beyond reads) when transactions commit, and SAP implements some aspects of the application framework we described in that application layer.

Stored procedures can improve performance and encapsulation by executing code within that data management system. For example, a scheduler could select intents to satisfy, and then schedule them using a single transaction running as stored procedure at an appropriate isolation level. This reduces pessimistic lock duration significantly, or reduces rollbacks if optimistic concurrency control is used in the DB.

Putting too much code within the DB might make it a bottleneck, and would reduce the flexibility of the application layer. Balancing database and application tier capabilities is an art with some challenging tradeoffs worth further consideration. We're interested in application (and application framework) specification paradigms that support multiple execution bindings to the application and database layer, with "best" binding selected by an optimizer.

4. OPERATIONAL BUSINESS INTELLIGENCE

In this section, we describe another example, an approach to operational business intelligence (OBI) using intent that we think may (appropriately) do a better job of delivering the actual intent of decision-makers better than some other approaches. We'll explain the problem, then talk about OBI with a single database, and finally look at OBI with multiple databases. That leads us to business process intent, which is discussed more generally in the section 5.

4.1 The operational business intelligence problem

When decision makers or other knowledge workers see reports generated from databases, they may detect problems in their businesses which they want to address. Something might be wrong that needs to be fixed, such as a (repeatedly) broken production line, or there might be a better way of handling an issue, such as a customer escalation. Business intelligence involves getting data from one or more databases so that decision makers can act on it in a timely way. At one time such reports were generated weekly or nightly, but decision makers increasingly want such information immediately, using current (or nearly current) data. Instead of requiring decision makers to ask the right questions, active databases may automatically alert people when unusual events (or unusual complex events, composed from other events, e.g., a sequence of events) occur. This allows people (or automated agents) to react quickly to these unusual events, handling them as soon as possible.

However, even if corrective actions are taken very quickly, the state of the database that the decision maker observed may have significant differences from the state of the database when action is taken. The production line might be under repair by a restart or other action that's underway. Another decision maker (human or automated) might have addressed the production line failure by diverting some other production line to deliver high priority products assigned to the failed production. An appropriate corrective action in the original state might be completely inappropriate in the current state, whether it's minutes, seconds or even fractions of seconds later.

Moreover, the state of the database and the state of the real world aren't necessarily in agreement. The observation that the production line was broken might be erroneous. Other production lines may have failed, so an appropriate action with only one production line down might no longer be appropriate. Even if the database state and the real world are in agreement, good and bad things might happen subsequently, such as a new production line starting or power going out in a factory. If the only information that is captured from the decision maker is the decision they requested, or worst yet, the data operations (and associated real world actions) they caused, then it seems impossible to deal with a situation different from what was expected when the decision was made.

The operational business intelligence (OBI) problem is determining how to use business intelligence to make operational business decisions that are appropriate for both the decision time and the current database state when the decision is applied. That is, decisions should continue to be applicable by application code in appropriate ways when the database state changes.

4.2 Intent and operational business intelligence for one database

Not surprisingly, we believe that intent is an excellent approach for OBI. If the intent of a decision is captured, not just a set of actions that effectuate that decision, then the intent optimization engine can try to meet that intent in any current (or future) database state.

For example, if a decision maker believes that a production line manufacturing a product for a sales order has failed, and product delivery is more important than was previously indicated, then the decision maker could increase the priority of that production job or of the sales order that generated that production job. If an objective function rather than priority is used to schedule machine production lines, then the decision maker could bump up the value of servicing this particular customer or this particular sales order.

If the production line has recovered, or if another decision maker has taken actions to ensure that this production order will complete, then the original decision maker's action will have no effect since their intent was already being met. The intent optimization engine might even be taking action to handle the production line failure already, based on the intent originally expressed for the sales order. Changing the priority of this production job might cause it to finish more quickly, or might have no effect.

Some decision makers might not understand detailed priorities and objective functions, and the impacts these could have on production line scheduling. It would be better if the user experience for decision makers were expressed in their own terms, where they could request (or require) that job constraints be met, possibly with coarse granularity priorities. Alternatively a rules-based approach could be used. Decision makers should also be able to view the intents that can be satisfied and the intents that cannot be satisfied with visualizations that match their roles and experiences. For example, if an intent cannot be met, then a summary visualization of higher priority intents might be visually presented showing revenue, responsible groups and priorities for those intents. For optimization algorithms, this can be achieved with explanation components that explain the constraints, and ideally also explain the contribution of various elements to the target function.[7]

4.3 Intent and operational business intelligence for multiple databases

Now let's consider the operational business intelligence problem when business intelligence data may come from multiple database sources, either via a data warehouse or via queries to these databases. For simplicity, we'll assume that there are two databases. At first glance, this may seem like the same problem that we considered in the previous section. If actions can be taken against both databases using distributed transactions with two-phase commit, then the problem is essentially the same as in the previous section. However, there are good reasons, such as performance and failure handling [Helland2007], for avoiding two-phase commit unless both databases are in the same management domain, and perhaps even when they are.

Let's assume that distributed commit is not acceptable, and the decision maker wants to take actions that affect both databases. Since we've assumed that two-phase commit isn't acceptable, we can't perform actions against the two databases atomically. Instead, however, we can perform actions similar to those in long running transactions and sagas [Gray1983, Garcia-Molina1987], where transactions are performed against both databases, and both must succeed for the saga process to complete successfully. A common example is planning an itinerary for a trip to a city, requiring both a roundtrip flight and a hotel, which are booked in different databases. If a traveler books a flight but can't get a hotel for those dates, then the flight will be cancelled, and the user may try to book the trip on different dates (or to a different city).

In an OBI situation, the traveler may see flights and hotels (and their prices) together in a warehouse, and may decide to book a particular flight and hotel. Using intent, the traveler could express date, price and hotel location constraints, as well as an objective function that trades off price with flight duration and hotel quality. The intent optimization engine could choose a flight and hotel for the traveler, and might book the flight, only to discover then the hotel no longer has rooms available for those dates. It might choose another hotel room, or might cancel the flight and select hotel and flight on another date. In some ways, this is similar to the calendar example from section 2, except that multiple databases are involved, so an application level business process is required, not just a single transaction.

If a room at a better hotel becomes available cheaply, then a satisfaction callback would be executed allowing the traveler to switch hotels. If staying at that hotel requires changing flight dates, then the change should only happen if the objective function (including flight change penalties) improves; the change in flight and hotel should be made carefully, so that the traveler doesn't release existing reservations until the new reservations are confirmed. Other changes, such as a hotel that closes down or a travel date change request made by the traveler, could also be addressed by a travel application based on the traveler's intent.

5. BUSINESS PROCESS INTENT

Most of the infrastructure and examples that we've discussed so far in this paper have involved use of intent for transactions, and the title of this paper is "Transactional Intent". However, intent can also be used for business processes, as illustrated by the Supply Chain Business Interactions and the Supply Network Interactions examples in section 2, as well as by the multiple database travel itinerary example in the previous section.

In this section, we first describe a practical basic approach to decomposing business process intent into transactional intents, and then mention some intriguing aspects of intent (internal, externalized and predictive intent) for multi-party processes.

[7] See, for example, APO Supply Network Planning, http://help.sap.com/saphelp_ewm70/helpdata/en/87/383e4229f1f83ae10000000a1550b0/content.htm

5.1 Intent decomposition

Addressing business process intent requires decomposing the business process and its intent into subparts (such as transactions), each of which has its own intent. Sometimes the decomposition into transactions and associated intents may seem clear from the definition of the process. For example, the travel itinerary example may be decomposed into multiple transactions in more than one way (get flight first and hotel second, or get hotel first and flight second). The transactional intents for the "flight, then hotel" decomposition would be "Find best flight meeting constraints" and then "Find best hotel meeting constraints, including added constraints imposed by the flight". But even this simple example raises questions: How were these decompositions identified? Do other plausible decompositions exist?[8] How were the transactions and transactional intents in the decomposition determined? In the example, how were "the added constraints imposed by the flight" identified?

Solving the general intent decomposition problem resembles solving a general bottom-up goal-directed artificial intelligence problem. We're not going to try to address such general automatic programming problems in this paper. Practical systems are more likely to have one or more decomposition patterns identified for each business process, describing the transactions and their intents, as well as other application framework aspects (e.g., satisfaction callbacks) described in section 2.

The transaction and intent decomposition patterns for the Supply Chain Business Interactions and Supply Network Interactions business process examples presented in section 2 are more complex and more flexible, involving not only transactions but also messages between parties, where each message expresses intent. However, there typically are standard decomposition patterns for these business processes; whenever the business process initiates a transaction, it can associate an intent with that transaction, while also identifying itself and its own intent.

5.2 Multi-party business processes

Processes which involve multiple parties introduce some new aspects because the two (or more) parties involved may not fully share their actual intents with each other. A purchaser may request a term sheet from a supplier, but its intent may be to pressure other suppliers to lower their bids. We can distinguish among the following types of intent in a two party interaction between purchaser and supplier:

- The purchaser's intent, known only by the purchaser, an internal intent.
- The supplier's intent, known only by the supplier, also an internal intent.
- The intent which the purchaser expresses in messages to the supplier, an externalized intent.
- The intent which the supplier expresses in messages to the purchaser, also an externalized intent.
- The intent which the purchaser infers from the supplier's past and current behavior, a predictive intent.
- The intent which the supplier infers from the purchaser's past and current behavior, also a predictive intent.

The supplier can act based on its own internal intent, the purchaser's externalized intent, and the predictive intent that the supplier has inferred from the purchaser's behavior; a similar statement can be made for the purchaser.

Behind the internal intent expressed in the purchaser and supplier software, there is also the intent of the human beings (if any) making decisions during the business process. Predictive intent analyzes externalized behavior to try to infer intent, but that intent could depend on the specific human beings making process decisions. One reason to store version histories including intents is to help with such predictions, although different people may have different intents and behaviors.

6. INTENT CONSIDERATIONS AND IMPLICATIONS

Section 2 described some applications that use (or could use) intent. Although many of the ideas in this paper have been in use in systems for many years, and others ideas, such as apologies, have been proposed before, the conceptual framework described in section 3 is novel, describing a high-level architecture for handling intent. Section 4 described use of intent for operational business intelligence, and section 5 discussed some aspects of process intent. In this section we present some additional considerations and implications, which we hope will lead to future work on transactional and business process intent.

6.1 Auditing, data mining and predictive analytics

Applications following the framework described in section 3 store a lot of data and metadata about intent, including data versions, intents and sub-intents, execution plans for meeting intent, satisfaction events/callbacks that induce re-planning, dependency graphs and other intent metadata. This information is valuable for many reasons; it can be used to explain decisions to users and partners, as well as for auditing and regulatory requirements. Moreover, it may be valuable for data mining and predictive analytics concerning decisions. For example, a supplier who frequently reneges on deliveries could be regarded as a risky choice for future critical orders, even if that supplier's terms are excellent. Predictions are useful even for single party processes, e.g., to predict whether a production run is likely to complete before its deadline during a busy time of the month. More generally, the higher level semantics expressed in intent may enable deeper data mining and richer predictions than can be made based only on data history.

[8] One such decomposition involves selecting multiple good flights and hotels independently in parallel, finding the best matching pair, and then reserving that flight and hotel, if possible, trying again if it's not.

6.2 Compensation and change

Utilizing the framework described in section 3, intent supports compensation when a submitted request can no longer be completed as originally planned. Intent can be the basis for a process exception management approach that is usable either within a single company or in B2B contexts, since it captures semantics and metadata which value and operation-based approaches omit. Instead of compensating for a failed plan and then building a new intent execution plan, these two steps could be correlated and combined, so that the old plan is modified rather than deleted and replaced. When it's workable, this approach could be relatively efficient. For example, dependency graph edges that exist in both old and new plans might be retained, rather than compensated for and then recreated.

6.3 Meta-intent

Intent may be expressed using imperative or declarative approaches (e.g., with declarative constraints). Intent optimization engines also may be written using imperative or declarative models, or a combination of both. Using a declarative engine makes it easier to specialize incremental or global intent optimization algorithms to meet specific circumstances. Let's consider a supply chain production scheduling example. One manufacturer may have particular policies and algorithms for Advanced Planning and Optimizer (APO) that are different than those for other manufacturers; these policies and algorithms capture that manufacturer's **meta-intent**, that is, his intent in optimizing the intents of his users.[9] (APO was described in section 2.3.)

Intent scheduling may also be polymorphic for a particular manufacturer, with different scheduling algorithms used for different classes of users or products. As usual, declarative encapsulation makes it easier to introduce, modify or replace such algorithms.

6.4 Eventual consistency

Intent can help deliver eventual consistency [Vogels2008] for replicated data based on ACID 2.0 (associative, commutative, idempotent, distributed) [Finkelstein2009, Helland2009]. For example, for scalability, availability and locality, there might be multiple sites that handle Advanced Planning and Optimizer (APO) requests and schedule jobs for the same factory, with job data (unique job names and job intents) replicated asynchronously among the sites. This approach could also be used for eventual consistency of multiple order entry sites, or for disconnected mobile calendars, which are much more likely uses than APO.

Assume that:

a) all job data eventually arrives at all sites, and

b) global scheduling is deterministic, based only on the set of distinct jobs, not on their order of arrival.

Then the job data at all sites will eventual converge, and the job schedules at the sites will eventually be consistent. (Timeliness is an important issue that we won't address here.) At each site, this approach to eventual consistency uses the application framework (with intent satisfaction events and callbacks) described in section 3. It does not require distributed transactions or complex replication protocols, only conditions a) and b).

6.5 Cooperating businesses and applications

Business process intent is a particularly important tool at the boundaries between worlds described by different applications and systems. For example, intent helps synchronize plans across customers and suppliers in a multi-tier supply chain, where satisfying an intent request from X to Y may depend on satisfying an intent request from Y to Z.

Intent could also be used to align the delivery schedules of multiple distribution centers run on different systems, so that they can cooperate to serve worldwide product demand.

6.6 Performance and usability

Used badly, intent may engender poor user experience and heavy system load. For example, if global optimization of a large APO system is performed every time a new purchase order is created, the performance load and rescheduling instabilities could be unreasonable. Like other optimization schemes, intent optimization needs to be properly calibrated; doing incremental optimization or partitioned optimization (over a related subset of orders) might be a better approach than continual global optimization. Global optimization could still be performed, but only infrequently, and its potentially disruptive results might be adopted only when they substantially superior to existing schedules.[10]

Other user experience issues include entering intent and writing applications using intent. We discussed ways that decision makers might enter intents in their own terms, perhaps using a "visual intent" interface. Programming applications using intent has many non-trivial aspects (such as dependency graphs) where tools, libraries and programming conventions would help. But we believe that systematic use of intent can make application programming easier, requiring less expert knowledge across different application domains.

7. RELATED WORK

The most closely related work that we know of is Helland's apology-oriented computing [Helland2007], which we've already discussed. This paper generalizes ideas in apology-oriented computing and proposes a framework for the generalization.

[9] Configuration and customization of an installation are metadata operations which implicitly or explicitly capture aspects of the installation's intent in processing requests, as opposed to the intent of a particular request.

[10] This destabilization problem may be a reason that calendar systems don't follow the approach suggested in section 2.2.

Field calls and commutative locks/escrow locks were mentioned in the introduction of this paper and are discussed in [Gray 1993], but intent is at a very different semantic level.

Semantic data types are discussed in papers including [Schwarz1984, Weihl1988]; intent uses semantics, but in a different way and at a different level.

There have been many papers on long running transactions; a number of early works are cited in [Gray1993], including Garcia Molina and Salem's work on sagas with compensating transactions [Garcia-Molina1987]. Our paper uses both ideas, long running transaction and compensation, but creates a novel framework using them.

Operational transform [Ellis1989] has been discussed recently because of its use in Google Wave. As with intent, the goal is to determine what transaction to perform in a changed state. Operational transform achieves this either because operations (such as appends to a thread) commute, or by inferring the operation from a state transformation. Intent does not require operation inference; it explicitly represents requested semantics, and it does this for a transaction, not just a single operation.

A number of papers (e.g., [Agrawal2009, Sadikov2010]) have examined the problem of meeting the intents of users performing web search, but although the word "intent" is used, these papers are addressing a very different problem than we are. Our predictive intent has an indirect connection to web search intent because both involve mining and prediction.

This document contains research concepts from SAP®, and is not intended to be binding upon SAP for any particular course of business, product strategy, and/or development. SAP assumes no responsibility for errors or omissions in this document. SAP does not warrant the accuracy or completeness of the information, text, graphics, links, or other items contained within this material.

8. REFERENCES

[Agrawal2009] Agrawal , R., Gollapudi, S., Halverson A. and Ieong, S. 2009. Diversifying Search Results, Proceedings of the Second ACM International Conference on Web Search and Data Mining.

[Balla2007] Balla, J. and Layer, F. 2007. Production Planning with SAP APO-PP/DS, Galileo Press, Boston, MA and Bonn Germany.

[Ellis1989] Ellis, C.A. and Gibbs, S.J. 1989. Concurrency Control in Groupware Systems. ACM SIGMOD Record 18 (2), p.399–407.

[Finkelstein2008] Finkelstein, S., Brendle, R., Hirsch, R., Jacobs, D., and Marquand, U. 2008. The SAP Transaction Model: Know Your Applications, presented (but not published) at ACM SIGMOD 2008 Product Day, Vancouver Canada.

[Finkelstein2009] Finkelstein, S., Brendle, R., and Jacobs, D., 2009. Principles for Inconsistency, Proc. CIDR, 2009.

[Garcia-Molina1987] Garcia-Molina, H. and Salem, K. 1987. Sagas, Proc. SIGMOD Conference 1987, p. 249-259.

[Gray1993] Gray, J. and Reuter, A. 1993. Transaction Processing: Concepts and Techniques, Morgan Kaufmann, San Mateo, CA.

[Hamady2009] Hamady, M. and Leitz, A. 2009. Supplier Collaboration with SAP SNC, Galileo Press, Boston, MA and Bonn Germany.

[Helland2007] Helland, P. 2007. Life Beyond Distributed Transactions, An Apostate's Opinion, Proc. CIDR 2007.

[Helland2009] Helland, P. and Campbell, D. 2009. Building on Quicksand, Proc. CIDR, 2009.

[Sadilov2010] Sadikov, E. , Madhavan, J., Wang , L., Halevy, A. 2010. Clustering Query Refinements by User Intent, Proc. of the 19th International Conference on World Wide Web.

[Schwarz1984] Schwarz, P.M. and Spector, A.Z. 1984. Synchronizing Shared Abstract Types, ACM Transactions on Computer Systems (TOCS), v.2 n.3, p.223-250.

[Vogels2008] Vogels, W. 2008. Eventually Consistent, ACM Queue, 6(6), p. 14.

[Weihl1988] Weihl,W. 1988. Commutativity-Based Concurrency Control for Abstract Data Types, IEEE Transactions on Computers, v.37 n.12, p.1488-1505.