In-memory Business Process Management

Sören Balko, Alistair Barros

Queensland University of Technology, Information Systems School, Services Sciences
Gardens Point Campus, 2 George Street, Brisbane, 4000, Queensland, Australia

Abstract

In-memory databases have become a mainstay of enterprise computing offering significant performance and scalability boosts for online analytical and (to a lesser extent) transactional processing as well as improved prospects for integration across different applications through an efficient shared database layer. Significant research and development has been undertaken over several years concerning data management considerations of in-memory databases. However, limited insights are available on the impacts of applications and their supportive middleware platforms and how they need to evolve to fully function through, and leverage, in-memory database capabilities. This paper provides a first, comprehensive exposition into how in-memory databases impact *Business Process Management*, as a mission-critical and exemplary model-driven integration and orchestration middleware. Through it, we argue that in-memory databases will render some prevalent uses of legacy BPM middleware obsolete, but also open up exciting possibilities for tighter application integration, better process automation performance and some entirely new BPM capabilities such as process-based application customization. To validate the feasibility of an in-memory BPM, we develop a surprisingly simple BPM runtime embedded into SAP HANA and providing for BPMN-based process automation capabilities.

Keywords: In-memory databases, NewSQL, Business Process Management, BPMN 2.0, SAP HANA

1. Introduction

The proliferation of always-connected mobile devices, social networks, the *Internet of Things*, and the uptake of Cloud-based deployment models poses a challenge for traditional, disk-based storage technology. This impacts enterprise applications, now incorporating these traditionally niche areas, leading to scalability issues[1] for unprecedented increases in data volume and transaction frequency.

Email addresses: Soeren.Balko@qut.edu.au (Sören Balko), Alistair.Barros@qut.edu.au (Alistair Barros)

[1] Affecting both transaction throughput and transaction processing times.

In response to these challenges, key developments have recently emerged in database technology. Significant among these are NoSQL databases [18] which improve scale-out characteristics compared relational databases, but which often trade consistency for availability and "partition tolerance" [6], thus falling short of ACID guarantees. Strong consistency is generally a highly desirable feature of enterprise applications [23, 19]. Following on from NoSQL databases, a number of long-standing relational databases concepts have been renovated, through "NewSQL" databases. While no narrow definition of NewSQL databases exist, some frequently found principles [25, 24] include: single-threadedness, which adopts the asynchronous and single-threaded design of Web application middleware such as *node.js* [27] and *nginx* [22]; a shared-nothing principle that avoids "singleton" components, often forming performance bottlenecks; and advanced concurrency control that abandons dynamic locking over techniques like MVCC, timestamp ordering, and clever global clock synchronization [9].

These design principles support better **scale-out characteristics**, where provisioning additional database nodes successfully counteracts performance degradation resulting from an increased load. Most authors position these architectural changes alongside empirical insights into how OLTP workloads and their underlying relational database schemas are structured. Even NewSQL databases only demonstrate favorable scale-out characteristics when supported by the actual database schema, the sharding, clustering, and replication strategies on top, and the actual transaction workload. For example, in [8, 24] the authors advise to employ single-sharded transactions (transactions that can be run on the data that is replicated to a single node, avoiding to invoke costly distributed transactions). Empirical insights into prevalent transaction and schema characteristics support the fact that many real-life OLTP workloads already rely on schemas that are trivially partitionable into shards supporting single-sharded transactions [25].

The aforementioned architectural "renovations" of NewSQL databases are complemented by placing OLTP databases (and if reasonably sized, also OLAP databases) in main memory. Avoiding slow access times of disk-based storage, main memory offers multiple orders of magnitude faster access times, avoids comparatively slow disk interfaces, and can benefit from vector processing instructions of modern CPUs to rapidly process large amounts of data in main memory. Oversized databases can use techniques such as compression or anti-caching [10] and still benefit from the performance advances of main memory storage. Transitioning from a disk-based (or Flash-based) storage to an in-memory storage essentially represents a **scale-up strategy**, benefiting the per-node performance of a distributed database. In-memory databases [17] have demonstrated their performance advances in early systems such as *Monet* [5], *C-Store* [5] and are now adopted by major software vendors, e.g. SAP ("HANA" [13, 14]), Oracle (Oracle Database 12c), Microsoft (SQL Server "Hekaton" [11]), IBM (DB2 "BLU" [21]) and others.

As the latency of database operations reduces, application interfaces such as ODBC and JDBC (copying query results into the application's address space, (de-)serializing data into (from) network packets, etc.) become a bottleneck for

end-to-end transaction turnaround times. Relocating data-intense portions of the application code into the in-memory database can avoid costly data copies and can greatly improve performance [24, 25].

As a result of applications migrating onto in-memory databases, some middleware services for application integration may become less relevant. For example, messaging services (such as *Enterprise Service Buses*) traditionally invoke public service interfaces of applications to transfer state and events between applications. When applications use the same database instance, the database as such may form a more efficient route to do so. While this is not a unique property of in-memory databases, two of their contributions make such a scenario more plausible:

1. The aforementioned "push-down" of data-intense application code into stored procedures within the (in-memory) database makes these low-level services accessible on the database level.
2. The aforementioned pledge for virtually unbounded scale-out characteristics of NewSQL databases makes them a suitable storage technology of a Cloud platform, where multiple applications and/or tenants can share the same underlying database instance. With the "lion's share" of a database's operational budget being spent on administration [26], this technological capability is augmented with a strong economic incentive. In fact, application vendors such as SAP already position their in-memory databases as the storage backend of their Cloud PaaS offerings.[2]

This paper provides a first and comprehensive exposition of the architectural implications for enterprise applications and, specifically, a key supportive technology in business process management [1], through in-memory databases. Our contribution is twofold. Firstly, we position the broader architectural implications of in-memory databases in the context of business process management and identify novel BPM capabilities that become possible through an in-memory database "underpinning". Secondly, by way of demonstrating the feasibility of an in-memory BPM, we present the building blocks of a BPM core runtime service for SAP HANA, where we provide a mapping of BPMN [20] artifacts to its "programming model"

The rest of this paper is organized as follows. In section 2, we provide general insights into how software application architectures are changing as a result of in-memory databases in conjunction with other trends underway. In section 3, we specifically look into how enterprise application integration (EAI) technologies such as business process management (BPM) are affected by the aforementioned architectural changes to enterprise applications, resulting from their migration to an in-memory database platform. In section 4, we give details of an in-memory BPM reference implementation based on SAP HANA and focused on process automation. In section 5, we summarize our contributions.

[2] http://hcp.sap.com/

2. Application Architecture Implications

Any database creates, effectively, a centralized technical hub, for applications inside an organization. Sharing a single database instance is economically attractive and is now made possible by the aforementioned scalability advances of modern in-memory databases[3]. Potentially, a single (distributed) in-memory database instance may even be shared among multiple tenants, such as companies participating in the same supply chain. That sharing model is particularly suitable for a Cloud deployment. For instance, a *Supply Chain Management* (SCM) application may be deployed onto a public Cloud and using an in-memory database that forms a part of the Cloud platform services. The SCM application may then allow the different tenants (i.e., the supplier(s) down to the recipient) to work with the same "business objects" that are stored in the (shared) underlying database instance. That capability lends itself to monitoring, alerting, and issue resolution scenarios, where the recipient may gain direct access to business objects (such as production and logistics schedules, etc.) of its suppliers. Altogether, a single, scalable in-memory database instance can provide both the persistence layer for multiple applications and tenants and also allow for efficient data exchanges and synchronization of state among these applications or tenants. In effect, introducing an in-memory database can lead to an effective reduction of application and tenant "silos".

In order for in-memory databases and their performance capabilities to be fully harnessed, applications need to run any data-intense operations inside them. Running application code within the address space of the in-memory database follows general recommendations for NewSQL databases [24] and is in stark contrast to long-standing database tuning practices. As a result, application code is "dispersed" among two stacks (application server and database system), effectively trading performance against a "clean" layering into a data access and an application layer. Consequentially, the conventional 3-tier architecture [12, 15] with a *physical* separation of the client, application, and database layer is diminished as applications fully migrate onto an in-memory database.

As a result of introducing scalable in-memory databases, Figure 1 illustrates how application architecture may evolve. In a traditional 3-tier architecture (on the left hand side), the application server is the sole container and runtime environment for application components. The (disk-based) database is merely used as a data storage and state synchronization facility. Disk-based databases frequently form the scalability bottleneck within the 3-tier architecture. As a mitigation action, application components incorporate certain data processing tasks. As a result, the load on the database (in terms of statement complexity and transaction processing times) is effectively reduced, such that general transaction throughput increases. However, this deliberate design choice requires application servers to perform costly data processing work locally, where

[3]Which are one incarnation of NewSQL databases.

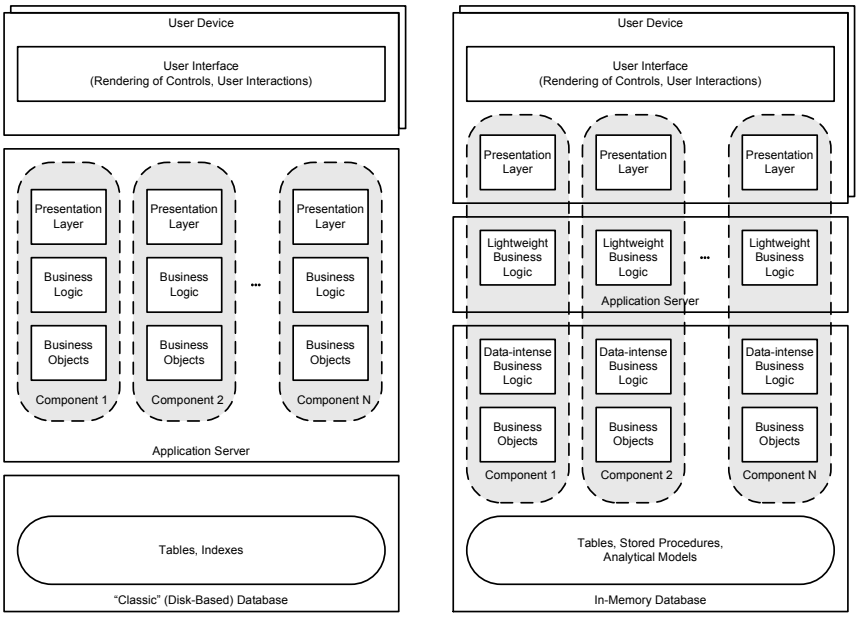

Figure 1: Evolution of the 3-tier Architecture

much data is frequently copied between the database and the application server. Adding further application server instances is the prevailing scale-out measure.

An alternative to the 3-tier architecture (right hand side of Figure 1) results from the programming model of in-memory databases and parallel trends through the progression of client-side technologies. At its core, applications on an in-memory database "push down" those portions of the code that process large amounts of data. This also includes any code which needs to read or write large amounts of data, such as end-of-period accounting procedures in an ERP system which need to roll up a large number of line items (such as individual expenses and invoices) to compute some totals. At the presentation layer, we already see application server functionality offset by powerful client-side technologies such as mobile devices and HTML5-compliant user agents providing presentation rendering. The application server is merely affected with provisioning of service interfaces (like REST endpoints).

Taken together, in-memory databases and modern user clients are serving to evolve application servers to a relatively slim layer of lightweight business logic and vertical capabilities (such as security; central configuration, administration and monitoring; software lifecycle management; etc.). While the long-term trajectory of this development is yet to unfold, we argue that the remaining functionality may not justify another stack layer (i.e., the application server)

and we expect the remainder of the application servers to also merge into a single, consolidated in-memory database. That consolidated platform should still expose logical layering and virtualization, where appropriate (e.g., when tenant isolation is mandatory).

3. In-memory Business Process Management

Business Process Management [1] is a prominent application integration technology. A business process model captures an execution order of activities through control flow dependencies, described in a flowchart-like notation. Business processes provide the basis of business aligned application coordination, by allowing different parts of applications to be composed together and coordinated through a process orchestration engine. Being able to *directly* inte-

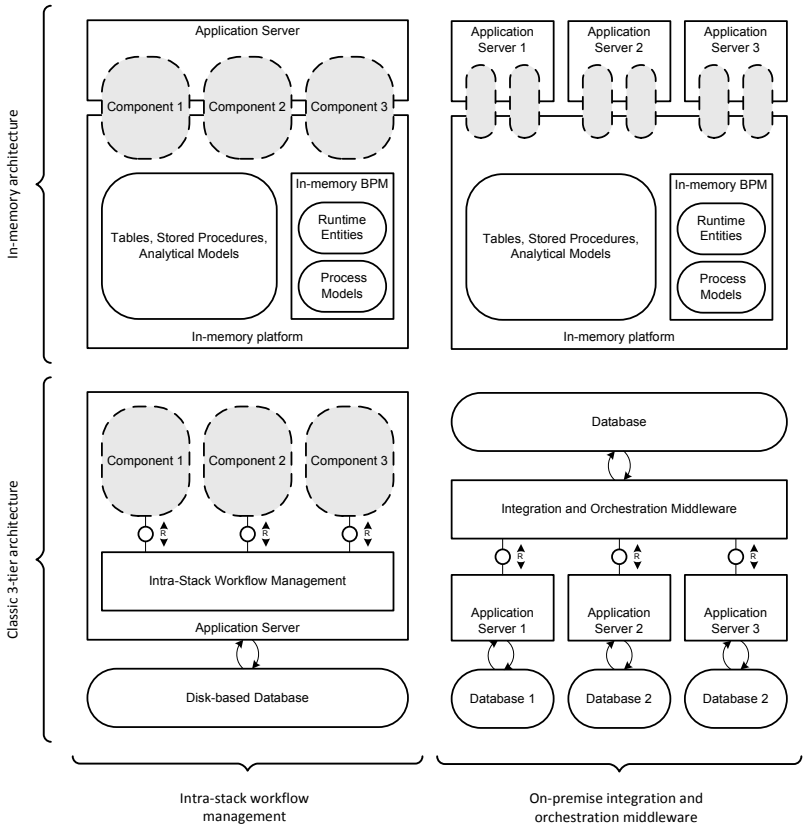

Figure 2: Deployment options for in-memory BPM systems

grate applications within the in-memory database poses the question as to how

the resulting new integration and orchestration capabilities (both for system-to-system and also human-to-system processes) are positioned against traditional enterprise application integration (EAI, [16]) technologies. Specifically looking at business process management (BPM) systems, we believe that an in-memory approach is superior to the traditional architecture and deployment of BPM systems as separate, self-contained systems. These separate BPM systems are constrained to integrating with applications through their existing public service interfaces. In contrast, an in-memory BPM system may access an application on various levels, starting at the raw database structures (tables, rows, attributes), over abstractions such as business objects (effectively corresponding to database queries/views that join rows from a number of tables), down to stored procedures (or other functional entities of the in-memory programming model) representing the "pushed down" (data-intense) code portions of an applications. Figure 2 illustrates the possible evolution of established BPM capabilities to a new in-memory foundation.

The bottom row of Figure 2 shows two existing high-level architectures and deployment models for BPM systems. The lower left diagram, illustrates an "intra-stack" workflow management capability which comes bundled with an application or platform. These intra-stack workflow management systems help automating and customizing functionality inside an application or platform, allowing to define "local" composites. Examples include SAP *Business Workflow*, Microsoft *Windows Workflow Foundation*, Oracle *Workflow embedded in E-Business Suite*, and Salesforce *Visual Process Manager*, noting that the application-centric variants such as SAP *Business Workflow* and Oracle *Workflow embedded in E-Business Suite* are being phased out by their vendors. In contrast, the platform-centric solutions such as Microsoft *Windows Workflow Foundation* (for the *.Net* platform) and Salesforce *Visual Process Manager* (for the *Force.com* Cloud platform) continue to be offered. The latter solutions have a wider reach beyond application boundaries, are consequently better suited to orchestrate tasks outside of a "monolithic" application stack and generally better positioned with regards to Cloud platforms.

The lower right diagram of Figure 2 depicts a simplified architecture of a separate integration and orchestration services, suitable for both an "on-premise" and Cloud deployment. Different to intra-stack workflow management solutions, integration and orchestration services are layered above individual applications and platforms and are suitable to integrate applications and systems from different vendors. Integration and orchestration services (such as IBM *WebSphere MQ*, Microsoft *BizTalk*, Amazon *Simple Queuing Services* etc.) are typically deployed onto their own physical (or virtualized) hardware infrastructure or are part of a Cloud platform's services. This is because these offerings are positioned as integration "hubs" exclusively using public service interfaces of the connected applications. BPM capabilities form a major part of the integration pipeline of integration and orchestration services. Customer-defined process models constitute complex integration semantics, orchestrating services provided by the connected applications and including human-centric activities.

Despite the maturity and rich functionality of the aforementioned types of

BPM systems, we argue that the underlying architectures do not fit the propositions of in-memory databases and may soon be rendered obsolete. Firstly, an intra-stack workflow management capability is nothing but another application component that can be migrated to an in-memory database (top-left of Figure 2), alongside other data-intense application components. Secondly, whenever different applications base on a single in-memory database instance, the case for an external integration services becomes obsolete, reason for that being the fact that an external services require "messages", which are to be exchanged in between two applications, to be passed from the sender application to the integration middleware, on to the receiver application. When both applications share a single in-memory database, this approach is a costly "detour", which is conceptually flawed in not using the most efficient communication route through the shared in-memory database instance. In effect, integration services such as business process management (BPM) and generally message-oriented middleware systems need to adapt and migrate into the in-memory database (top-right of Figure 2).

We believe that the use of shared in-memory database instances underneath applications yields superior ways to integrate applications and their tenants. We retain a model-driven approach to define integration scenarios, such as business processes. The architectural changes and technical capabilities made possible by in-memory databases will give BPM users more choice as to how said integration can be accomplished. The upper row of Figure 2 illustrates how we believe business process management (in its two aforementioned generic embodiments) to be affected by the proliferation of in-memory database for enterprise applications. In either case, we propose to embed core BPM capabilities into the in-memory database.

Our approach offers richer integration capabilities and better runtime performance. This is because any external cross-application communication is replaced with local communication within the in-memory database instance. While an in-memory database instance will typically still constitute a distributed database, the sharing, clustering, and replication strategies can be chosen in a way that each step a single business process instance "finds" all of its data (such as the accessed business objects) on a single node of the distributed database.

We partly follow the established trend of applications adapting to the in-memory programming model. That is, both major portions of a BPM runtime system and the business processes as such are implemented as (or automatically compiled into) in-memory database entities (such as table definitions, stored procedures, etc.). We further propose to broaden the spectrum of interfacing options between applications and business processes, where we generally see application integration enhanced in a number of ways:

1. In terms of service or data integration (or mixtures thereof) where a business process may access applications through code (e.g., by invoking their public service interfaces or stored procedures representing "pushed-down" application code that resides within the in-memory datavase) or through data (e.g., by directly querying and updating data that is "owned" by

some application).

2. In terms of data granularity (e.g., single attributes, database tables, business objects, business object hierarchies, etc.) where an in-memory BPM service can interface with applications by querying or updating those application artifacts, offering more flexibility than merely integrating applications through their public service interfaces.
3. In terms of request directions (i.e., "push", "pull") where a process may either actively trigger an application functionality or subscribe to application events and be notified upon their occurrence.
4. In terms of different consistency models, where a business process may be synchronously coupled to application transactions, be asynchronously (yet, reliably) de-coupled from application transactions, or be asynchronously de-coupled, yet *eventually* consistent [28] by means of some asynchronous consistency protocol.

Interfacing application data entities by other means than invoking pre-planned public service interfaces poses the risk of violating contracts and protocols that are typically enforced by those service interfaces. While this problem is real and may potentially "corrupt" persistent application state, we propose the following mitigation actions:

- Adhering to the constraints that are normally enforced by a service interface at runtime may often be relegated to a design-time problem. That is, a process modeling tool for an in-memory BPM service should not simply give modelers access to the database catalog (i.e., show plain tables) but integrate with a business object repository, which is part of many business applications (such as SAP's ERP stack). By relying on a higher-level "semantic" type system, the description of the listed business objects already constitute a "safe" CRUD interface to their state. Moreover, some (alas, not all) business objects are associated to a state machine model, which defines discrete states as combinations of a business object's attribute values and valid state transitions on top. An in-memory BPM runtime may enforce these constraints by compiling the state machine into database queries that are run from within the stored procedures which we generate for the actual process steps.

- Public service interfaces of business applications can be roughly subdivided into a RPC-style interfaces (such as SOAP-based Web services) and CRUD-style interfaces (such as RESTful Web services). Recent trends have de-emphasized RPC-style interfaces in favor of CRUD-style interfaces. Besides other differences, CRUD-style interfaces are closely angled around a "public" data domain model. That is, invoking a CRUD-style service is conceptually very similar to directly querying, updating, or deleting (portions of) a business object. Adapting applications exposing CRUD-style service interfaces to our proposed approach of interfacing business objects directly on the database level, may, in fact merely leave out the "overhead" of the Web services stack, but essentially perform the

same actions on the database. Service contracts (such as the aforementioned state machine model) can, in fact, be part of the "pushed down" application code, represented as database constraints, trigger definitions, etc.

- Finally, our approach does not dis-allow invoking applications through their public service interfaces, when appropriate. That naturally includes interfacing with *external* applications and services, which do not share the same database instance.

With those mitigations in place, we believe that the benefits of interfacing with application on the storage level often outweighs the disadvantage of suffering a potential loss of encapsulation. Beyond core performance reasons, integration middleware may benefit in other ways from the migration into an in-memory database. By way of BPM, in-memory technology enables a number of principal capabilities:

Deep and flexible application integration: A BPM system that resides within the same in-memory database as the applications, which it integrates has a far greater reach into application entities such as business objects and events. Being able to interface with an application using the different perspectives (code or data), levels of abstraction, request directions, and consistency models offers powerful ways to deeply integrate a business process with applications.

End-to-end model driven process foundation: At the time when application service interfaces are designed and implemented, future integration requirements are difficult to anticipate. Hence, the use of external composition processes is often limited in practical settings, where augmenting application interfaces is a costly governance exercise with long execution delays. Our approach overcomes this limitation by not exclusively relying on public service interfaces but integrating applications through a variety of other (implicit) interfaces.

IT landscape consolidation: Replacing separate, stand-alone integration servers (such as stand-alone BPMS) with in-memory integration platform services constitutes an IT landscape consolidation where the number of distinct systems in an IT landscape is reduced. The latter goes along with a measurable cost reduction (TCO) for system maintenance.

Architectural simplification and reuse: From a BPMS vendor perspective, there is a substantial potential of reusing existing in-memory database features to build a BPM runtime service. In our prototypical implementation, we were relieved of most dealings with complex aspects such as concurrency control and data consistency for the state making up a process instance; Cloud "readiness" (scale-out); robustness and failover guarantees; security (authentication and authorization); recovery and backup; data transformations and rich expression evaluation capabilities; event-based

action triggering; software and content lifecycle management etc. The enormous savings in terms of development efforts also help reducing implementation risks and benefit the overall BPM system's early maturity. We will re-iterate on some of these benefits in more detail in Section 4.

Intra-process analytics: Being embedded in an in-memory database with built-in OLAP capabilities (as is the case for SAP HANA), business processes could integrate the aforementioned high-performance, flexible "live" data analytics features. This principal capability can be employed in many scenarios like task routing based on complex data analytics, where the further process execution depends on a situational analysis of the current data basis. In an extended scenario, the underlying data analytics could actually even be altered by a process end user at runtime, enabling ad-hoc process flexibility.

Multi-paradigm support: Business processes are not limited to control-flow paradigms where a process model effectively defines the sequencing of process steps. Equally important, other paradigms (like business rules evaluation, complex event processing, and data transformations), augment or even supersede the control flow perspective of a process model. For example, this includes declarative approaches for constraint-driven processes. The in-memory process model actually makes it possible to simultaneously support and merge a number of model-driven "execution paradigms".

The aforementioned virtues of providing a core BPM system inside an in-memory database (henceforth conveniently named "in-memory BPM") augment existing BPM middleware capabilities, where these are still applicable in the context of the new architecture.

4. Approach

This section outlines the principles of a BPM runtime component running within an in-memory database and suitable to run BPMN 2.0 [20] based process models. We introduce a basic state model for process instances, illustrate the mapping principle for BPMN-based process models to SQL and stored procedures, and give the concrete mappings for a number of selected BPMN entities. Some familiarity with BPMN 2.0 is required in reading this section, which constitutes a conceptual proof-of-feasibility and leaves the necessary performance and scalability evaluations to be covered by future work.

Supporting stringent consistency requirements of business applications, in-memory databases (as one "incarnation" of NewSQL databases) provide for full consistency, supporting transactions with ACID guarantees. Throughout this paper, we will exclusively look at the aforementioned class of relational in-memory databases, exemplified by SAP HANA which provide for the following fundamentals:

- ACID style transactions with (at least) a "read committed" isolation level

- SQL interface for data definition, querying, and manipulation
- Procedural extensions ("stored procedures") supporting control flow constructs like (recursive) procedure invocation, looping, and conditional branching ("if-then-else") and database triggers for transparent nested execution of user-defined statements
- Scalability advances inherited from architectural changes introduced with NewSQL databases (c.f. Section 1) and based on schema, transactional workload, sharding, clustering, and replication strategies that support single-sharded transactions [8, 24].
- Per-node performance increase ("scale-up") resulting from in-memory data representation (possibly augmented by anti-caching strategies)

Besides the latter two criteria, popular disk-based relational databases (including open-source databases such as *MySQL* and *PostgreSQL*) satisfy our requirements. In fact, our approach is functionally compatible with those systems, but nonetheless ill-suited for pre-NewSQL relational databases for two reasons:

1. For one, the recent scalability advances of relational databases can partly be attributed to the architectural innovations of NewSQL databases (c.f. Section 1). Some (yet not all) of these innovations were transferred to other relational databases, most prominently including an in-memory primary data representation.
2. Secondly, the 3-tier programming model that is traditionally exercised for applications on top of relational databases does not encourage relaying application code into the database's address space (e.g., as stored procedures). In contrast, NewSQL (and in-memory) databases promote using stored procedures for best performance. As a result, pre-NewSQL databases would not host application code and "only" offer access to database tables, limiting the interfacing options at the storage level for a BPM runtime.

Extended functionality of our approach may even use "advanced" functionality of certain in-memory databases such as analytics (OLAP) capabilities on "live" data.

Our approach fundamentally embraces an event-condition-action (ECA) paradigm to automate business processes [3, 7] within an in-memory database. An event is a database state change, reflecting records in a table being inserted, updated, or deleted. Any such event is generally suitable to trigger process steps. Condition expressions can aggregate multiple events, which can jointly trigger a process step. Formally, these conditions are expressions in first order logic (FOL), reasoning about the database state. For instance, the (simplified) triggering condition for BPMN's *Synchronizing Parallel Gateway* is as follows:

$$\exists p.p \in P \wedge \exists t_1.t_1 \in T \wedge \exists t_2.t_2 \in T$$
$$\wedge t_1[processId] = t_2[processId] = p \wedge t_1[pos] = 1 \wedge t_2[pos] = 2$$

In relational database parlance, the expression requires a tuple p to exist in some relation P and two tuples t_1, t_2 in another relation T. P and T would correspond to to tables holding records representing individual process instances and process tokens, respectively. Process tokens reference a process instance through a foreign key attribute "processId", thus associating tokens to a particular *owning* process instance. Tokens further have a "position" attribute, storing the current position of that token within the control flow graph. In the given example, the synchronizing join gateway had two inbound edges labeled 1 and 2 and a token being positioned on either edge in front of that gateway had the corresponding values of its "position" attribute. This FOL expression could be easily converted into a SQL query like that:

```
SELECT
  "..." AS artifactId, "AND_JOIN" AS artifactType, p.ID AS processId,
  t1.ID AS tokenId, "tokenId2=" || t2.ID AS customParameters
FROM
  PROCESSES AS p, TOKENS AS t1, TOKENS AS t2
WHERE
  t1.POSITION = 1 AND t1.PROCESS_ID = p.ID AND
  t2.POSITION = 2 AND t2.PROCESS_ID = p.ID
```

Notice that the result tuples contain two "constant" attributes ("artifactId" and "artifactType"), identifying the unique model identifier of the gateway artifact and the type of the artifact as such. We will later explain why we need this extra information to be part of the query's result set. We further make an attribute "customParameters" part of the result set. This attribute is a character string using an URL parameter encoding scheme to accommodate any number of extra key-value pairs. In the example above, we use it to pass on the primary key of the second token.

The database state changes forming the events result from transactions being run by database clients. These clients can be any application, such as an ERP stack which updates its business objects. Alternatively, database state changes may also result from transactions that are internal to the database, such as process steps which, in turn, perform updates on the database. These updates may both affect the state representing the process instance as such (like when advancing a token of the process instance) and also update external state such as business objects which are "owned" by other application.

We map process models to a set of event-condition-action (ECA) rules, where the events and conditions are represented as database queries and an action comprises one or multiple stored procedures. Each stored procedure is, in turn, a sequence of database queries and update statements interspersed with control flow constructs (loops, conditional branching, invocation of other procedures, etc.). For example, the "action" for the *Synchronizing Parallel Gateway* from above is as simple as this:

```
// parse the "customParameters" attribute into a variable "tokenId2"
DELETE FROM TOKENS WHERE ID = tokenId1;
```

```
UPDATE TOKENS SET POSITION = 3 WHERE ID = tokenId2;
```

That is, the two tokens t_1, t_2 from the gateway's inbound edges are conceptually "merged" into a single token for the outbound edge by deleting the first token and setting the second token's "position" attribute to the label of the outbound edge. Other process artifacts have more complex actions, where (besides altering the internal process state), other database entities may be affected. For instance, a process artifact may well alter business objects of some external application by directly updating the corresponding database tables. For the action to be triggered, it needs to be technically related to the event and condition of the associated ECA rule. Like in case of the synchronizing join gateway example, any time after a state change to the processes or tokens tables when the query corresponding to the condition yields a non-empty result set, the corresponding action will be executed for the respective result tuples.

Mapping BPMN entities to ECA rules which are mapped to supporting database structures and stored procedures benefits the versatility of our approach. Conceptually, other modeling paradigms that have a mapping to ECA rules are also covered by our approach. From a BPM perspective, this is particularly rewarding for "adjacent" paradigms, which are frequently used in conjunction (or as part of) business processes, including: (1) business rules definitions (like decision tables, if-then-else cascades, etc.), (2) business activity monitoring (usually accomplished by ways of complex event (stream) processing) and (3) event correlation (as used in BPMN's message-triggered *Intermediate Catch Events*). We leave further details on other paradigms as subject to future work.

4.1. Synchronous and Asynchronous artifacts

Depending on the technical characteristics of the process at hand, two ways of triggering an action upon an event may apply:

- For *synchronous artifacts*, we group these artifacts into a single database transaction. Within that transaction we recurrently test the triggering conditions of all the assembled process steps. When the query that corresponds to a triggering condition of a process step returns a non-empty result set, we instantiate and run the corresponding action (a stored procedure). Only when all queries, which collectively represent the conditions of the assembled process steps return an empty result set, the transaction is complete.

- For *asynchronous artifacts*, we split the process model into "synchronous segments" where each synchronous segment is run in the aforementioned manner for synchronous artifacts. The synchronous segments are themselves related to database triggers which invoke the stored procedure for the synchronous segment in a separate transaction.

We will subsequently explain the two concepts in detail and explain the mapping of a BPMN-based process model by means of some examples.

In the simple-most case, a process model exclusively comprises synchronous artifacts. Synchronous artifacts are those process steps (i.e., gateway, activity, event) that can run without "blocking" the process by waiting for an asynchronously incoming external event. An incoming event denotes an external database state change on which a process step depends and which is not caused by some upstream process step. For example, BPMN's (message-triggered) *Intermediate Catch Events* block a process instance until a "message" is received. In technical terms, the receipt of that message denotes an external event. In contrast, a *Synchronizing Parallel Gateway* merely depends on events (database state changes), which are caused by upstream process steps. In detail, the triggering condition of the *Synchronizing Parallel Gateway* refers to database state changes (creating the process instance and putting the process tokens onto the gateway's inbound edges), which are exclusively caused by upstream process steps. In effect, the *Synchronizing Parallel Gateway* is classified as a synchronous artifact, whereas the *Intermediate Message Event* is an asynchronous artifact.

Please note that our classification into synchronous and asynchronous artifacts merely reflects the ability to group artifacts into a single transaction (or not). Other concerns (notably those affected with message exchange patterns or control flow parallelism) may result in different classifications, which are irrelevant to our approach. That is, we can group multiple synchronous artifacts, which are interconnected through control flow edges into a single transaction. An asynchronous artifact may only be the first process step in that transaction. The table below classifies some key BPMN artifacts as being asynchronous or synchronous: Besides purely synchronous and purely asynchronous artifacts, a

	Synchronous	Asynchronous	Sync-Async
Events	*End (Throw) Event*, *Intermediate Throw Event*	*Start (Catch) Event*, *Intermediate Catch Event*	
Gateways	*Exclusive Gateway*, *Inclusive Gateway*, *Parallel Gateway*, *Complex Gateway*	*Event-based Gateway*, *Parallel Event-based Gateway*	
Activities	*Send Task*, *Service Task*, *Script Task*, *Business Rule Task*, *Call Activity*	*Receive Task*	*User Task*, *Manual Task*, *Call Activity*

Table 1: Synchronous and asynchronous process artifacts

third "sync-async" category classifies some BPMN artifacts as "hybrids" having a leading synchronous and a trailing asynchronous part. Technically, these artifacts synchronously perform some initial work. Only when this initial work is completed, sync-async artifacts "block" and wait for an external event before

commencing with the asynchronous part.

For example, a *User Task* wraps the technical interaction with a task management software (through protocols like *WS-HumanTask* [2]), which is responsible for serving the tasks to end users. Human interactions are by definition asynchronous, i.e., the *User Task* artifact blocks until a human task processor manually picks up the task and subsequently marks it as "completed". Initially dispatching the task specification (comprising data to populate forms, instructions on who is eligible/excluded to/from processing the tasks, completion deadlines, etc.) to the task management happens synchronously before the *User Task* artifact blocks by waiting for the end user to complete the task.

4.2. Synchronous Artifacts Mapping

For a given process (or process fragment) exclusively comprising synchronous process artifacts, we can groups these process artifacts into a single transaction. The corresponding mapping algorithm is devised into three stages, being (1) the **preprocessing stage** where we label all control flow connectors ("edges") with an integer identifier, the (2) **mapping stage** where we map each artifact into a query and a stored procedure, collectively representing an ECA rule, and (3) the **assembly stage** where we assemble the queries and stored procedures from the mapping stage into a database view and a global stored procedure polling that view and invoking the fitting stored procedures from the mapping stage. The mapping algorithm generates SQL statements and stored procedure definitions, which interact with a minimal BPM runtime system. That BPM runtime system is itself a set of tables and stored procedure definitions that jointly provide capabilities that are shared between all processes. Different to

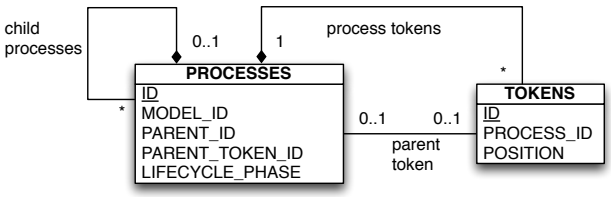

Figure 3: Minimal database schema for process runtime

most applications, BPM systems have a central notion of "process instances", being instantiated process models (represented by the `MODEL_ID` attribute) and defining the lifecycle of a single coherent "case". The `LIFECYCLE_PHASE` attribute may assume distinct values such as "running", "suspended", "canceled", "completed", "archived", etc. Process instances can be "nested" where parent processes can transitively invoke child processes. A process instance record, thus maintains a null-able `PARENT_ID` foreign key denoting its parent process instance (if any). Child processes also keep a null-able `PARENT_TOKEN_ID` foreign key attribute to the token of the parent process instance which has triggered the sub-process from within the parent process. Remembering that parent token is

important to correctly resume the parent process once the child process is complete. Within a single process instance, one or many tokens may exist, where each token simultaneously represents a parallel "thread of control" and a position within the control flow definition of the underlying process model. Tokens exclusively belong to a single process instance (denoted through its `PROCESS_ID` foreign key).

We further need a stored procedure `DECODE_PARAMETERS` and table type definition `PARAMETER_TYPE` in order to deal with the URL-encoded "customParameters" attribute. `DECODE_PARAMETERS` is passed a parameter string, listing a sequence of name-value pairs, following the syntax of URL request parameters

```
name1=value1&name2=value2&...
```

which it parses into a table instance of `PARAMETER_TYPE`:

```
CREATE TYPE BPM.PARAMETER_TYPE AS TABLE (
  NAME VARCHAR(50) PRIMARY KEY, VALUE VARCHAR(100));
```

Same as the `PROCESSES` and `TOKENS` tables, `PARAMETER_TYPE` is assigned to a schema "BPM", which hosts the generic BPM runtime database definitions. Parsing a "customParameters" string into an instance of the `PARAMETER_TYPE` table type is trivial such that we omit the `DECODE_PARAMETERS` stored procedure definition.

4.2.1. Preprocessing Stage

The preprocessing stage initially traverses the given process model to normalize the model and label the control flow connectors. The traversal starts at the process model's entry point(s), being those artifacts which do not have any inbound connectors. The pre-processing algorithm will then perform the following operations:

Connector labeling: Assign each control flow connector a label and make that label available in a lookup structure for the subsequent compiler stages. The connector labels will generally be unique, except for two cases:

- For (converging) *Exclusive Gateways* ("XOR joins"), the gateway's inbound connectors will all be labeled with the gateway's (single) outbound connector's label.
- For *Event-based Gateways* ("deferred choice"), the gateway's outbound connectors will all be labeled with the gateway's (single) inbound connector label.

Inbound connector normalization: Find all artifacts having more than a single inbound control flow connector that are not converging gateways. For each of these artifacts, insert a (converging) *Inclusive Gateway* ("OR Join") in front of that artifact.

Outbound connector normalization: Find all artifacts having more than a single outbound control flow connector that are not diverging gateways. For each of these artifacts, insert a (diverging) *Parallel Gateway* ("AND Split") behind that artifact.

We refrain from giving the algorithms to perform the aforementioned steps and refer to an example instead. Figure 4 shows a simple process model, comprising

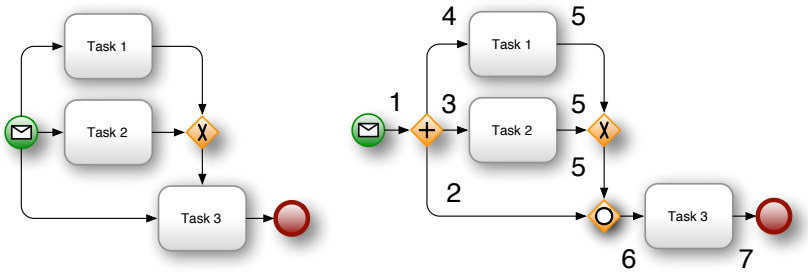

Figure 4: Process model pre-processing

a *Start Event*, an *End Event*, a converging *Exclusive Gateway*, and three *Tasks*. The preprocessing stage inserts a diverging *Parallel Gateway* behind the *Start Event* and a converging *Inclusive Gateway* in front of "Task 3". The control flow connectors are labeled with integer identifiers. These labels are generally unique except for the inbound and outbound connectors of the converging Exclusive Gateway which are all labeled "5".

4.2.2. Mapping Stage

The mapping stage re-visits all process artifacts and generates fragments of SQL and stored procedures, which are later assembled into a deployable file. Each artifact is mapped into (1) a query representing the event and condition part and (2) a stored procedure, which represents the action part. We subsequently give the mapping instructions for a a number of fundamental BPMN artifacts in order of increasing complexity. We deliberately refrain from giving mapping rules for the complete set of all existing BPMN artifacts, which beyond the scope of this paper.

Plain Tasks. For example, a simple "no-op" Task artifact which merely forwards a token from its inbound control flow connector to its outbound control flow connector is mapped onto the following query and stored procedure pair:

```
SELECT
 "..." AS artifactId, "TASK" AS artifactType, p.ID AS processId,
 t.ID AS tokenId, NULL AS customParameters
FROM PROCESSES AS p, TOKENS AS t
WHERE p.LIFECYCLE_PHASE = "running" AND t.PROCESS_ID = p.ID AND
 t.POSITION = <identifier of inbound connector>
```

```
CREATE PROCEDURE MYPROCESS.<artifactId> (
 IN processId INTEGER, IN tokenId INTEGER,
 IN parameters BPM.PARAMETER_TYPE)
LANGUAGE SQLSCRIPT AS
BEGIN
UPDATE TOKENS SET POSITION = <outbound connector label>
  WHERE ID = :tokenId;
END;
```

The query returns a result set of all tokens and associated process instances, where the token is directly "in front of" the task. Notice that we do not use the "customParameters" attribute in this case.

The stored procedure, representing the action part of the corresponding rule merely forwards the token to the task artifact's outbound connector by updating its "position" attribute. Tasks and activities performing any work (such as *Script Tasks*, executing a scripted sequence of program code) would naturally need to include additional DML statements into their stored procedures. For example, a *Script Task*'s script code could be cross-compiled into stored procedure code which was simply pasted into the stored procedure.

Forking. A diverging *Parallel Gateway* forks multiple branches by putting tokens on all of its outbound connectors. The query representing the triggering condition is identical to the one shown above (for Task artifacts). The difference lies in the action part where we need to insert additional token records for the gateway's 2nd, 3rd, etc. outbound connector:

```
CREATE PROCEDURE MYPROCESS.<artifactId> (
 IN processId INTEGER, IN tokenId INTEGER,
 IN parameters BPM.PARAMETER_TYPE)
LANGUAGE SQLSCRIPT AS
BEGIN
UPDATE TOKENS SET POSITON = <1st outbound connector label>
 WHERE ID = :tokenId;

INSERT INTO
 TOKENS(ID, PROCESS_ID, POSITION) VALUES(<new unique id>, :processId,
  <label of 2nd outbound connector>);
END;
```

This simplicity of spawning additional threads by merely creating further records in the `TOKENS` table is one of the conceptual strengths of our approach. Both the labels of the gateway's first and second outbound connectors are hard coded into the generated stored procedure code. The primary key of the newly inserted token record must be taken from a sequence or UID generator function (omitted for brevity).

Conditional Branching. An diverging *Exclusive Gateway* needs to evaluate user-defined branching conditions to decide onto which outbound connector a token is to be placed, where the branching conditions are "black boxes" to our approach. As a very simple example, suppose the gateway had three outbound connectors, of which the third was a "default branch" which is activated if none of the others' conditions evaluate to "true". The gateway's responsibility is to implement a staged purchase approval scenario where, depending on the monetary amount of some purchase order, the control flow was routed to different downstream branches.

```
SELECT
 "..." AS artifactId, "XOR SPLIT" AS artifactType,
 p.ID AS processId, t.ID AS tokenId,
 "outboundLabel=" || (
  (SELECT <branch1> FROM PURCHASE_ORDERS WHERE VALUE<100) UNION
  (SELECT <branch2> FROM PURCHASE_ORDERS WHERE VALUE>=100 AND VALUE<1000) UNION
  (SELECT <branch3> FROM PURCHASE_ORDERS WHERE NOT((VALUE<100) OR
   (VALUE>=100 AND VALUE<1000))
 ) AS customParameters
FROM PROCESSES AS p, TOKENS AS t
WHERE p.LIFECYCLE_PHASE = ''running'' AND t.PROCESS_ID = p.ID AND
 t.POSITION = <inbound connector label>
```

We embed the (mutually exclusive) routing conditions of the three branches (two branches with explicit conditions and one "default" branch) into the condition query. That is, the `customParameters` is used to store *outboundLabel* parameter that is set to the label of the *Exclusive Gateway's* outbound connector ("branch1", "branch2", "branch3") that gets activated. In effect, the resulting stored procedure merely had to use the returned *outboundLabel* parameter to update the token's `POSITION` attribute:

```
CREATE PROCEDURE MYPROCESS.<artifactId> (
 IN processId INTEGER, IN tokenId INTEGER,
 IN parameters BPM.PARAMETER_TYPE)
LANGUAGE SQLSCRIPT AS
 outboundLabel INTEGER;
BEGIN
SELECT VALUE INTO outboundLabel FROM :parameters WHERE NAME=‘outboundLabel’;
UPDATE TOKENS SET POSITON = :outboundLabel WHERE ID = :tokenId;
END;
```

Complex Synchronization. As a final example, the converging *Inclusive Gateway's* ("OR Join") complex synchronization behavior can also be represented in an equally simple manner. An OR Join must pass a token to its outbound connector *iff* it has a token(s) on at least one of its inbound connectors and for none of the inbound connectors having no tokens, there is a token further upstream that can reach that inbound connector. When triggered, the gateway

will then remove a single token each of those inbound connectors that have a token and put a single token onto its outbound connector.

Various alternatives have been proposed to efficiently implement OR joins in workflow engines, e.g. [30, 29]. Our approach is based on a variant that was conceived for the SAP *NetWeaver BPM* runtime [4]. The core idea is to determine for each of the gateway's inbound connectors the set of upstream control flow connectors. In case of the example process from Figure 4, the OR join's first (upper) inbound connector's (labeled 5) upstream connectors were the ones labeled with $\{1, 3, 4\}$.

Please note that we exclude 5 from that set as it is also the label of the inbound connector itself and does not count as "upstream" connector label. The second (lower) inbound connector's (labeled 2) upstream connectors were these labeled $\{1\}$. We first determine a result set of "candidates" being combinations of tokens at the gateway's inbound edges which may (potentially) jointly trigger the gateway. In our example scenario, these token candidates are returned by querying a `CANDIDATES` view as defined below:

```
CREATE VIEW CANDIDATES AS
 SELECT EDGE1.PID AS processId, EDGE1.TID AS tokenId1, EDGE2.TID AS tokenId2
 FROM
  (SELECT PROCESSES.ID AS processId, TOKENS.ID AS tokenId
   FROM (TOKENS JOIN PROCESSES ON TOKENS.PROCESS_ID=PROCESSES.ID)
   WHERE TOKENS.POSITION=2) AS EDGE1
  FULL OUTER JOIN
  (SELECT PROCESSES.ID AS processId, TOKENS.ID AS tokenId
   FROM (TOKENS JOIN PROCESSES ON TOKENS.PROCESS_ID=PROCESSES.ID)
   WHERE TOKENS.POSITION=5) AS EDGE2
  ON EDGE1.processId=EDGE2.processId
```

The view's query performs a full outer join on (`TOKENS`, `PROCESSES`) pairs that constitute process tokens residing on different inbound connectors of the gateway and their associated process instances. The gateway may synchronize varying numbers of tokens at a time, depending on the available tokens on the gateway's inbound connectors. In case of two inbound connectors, it may either synchronize one or two tokens at a time. The exact combination of inbound connectors from which to consume tokens is only known at runtime, where for N inbound connectors $O(2^N)$ combinations of inbound connectors having tokens exist. Our approach covers all of these combinations using $O(N)$ operations. Despite using a full outer join to form a tuple of candidate tokens on the gateway's inbound connectors, the nested selection (`WHERE TOKENS.POSITION=...`) creates small input tables on either side of the join operator, keeping the runtime cost low.

Even when candidate tokens are present, an OR join must not trigger if one of its inhibiting conditions holds. The `INHIBITORS` view queries the primary keys of all inhibited process instances where the OR join cannot currently synchronize any candidate tokens. An OR join is inhibited if for those inbound connectors that do not currently carry a candidate token, there are upstream tokens which can still potentially reach that inbound connector.

```
CREATE VIEW INHIBITORS AS
 SELECT DISTINCT INHIBITED.ID AS processId
 FROM TOKEN JOIN PROCESSES AS INHIBITED ON TOKEN.PROCESS_ID=INHIBITED.ID
 WHERE (TOKEN.POSITION IN (4, 3, 1) AND
  NOT EXISTS (SELECT *
   FROM TOKEN JOIN PROCESSES AS ENABLED ON TOKEN.PROCESS_ID=ENABLED.ID
   WHERE TOKEN.POSITION=5 AND INHIBITED.ID=ENABLED.ID
  )) OR (
  TOKEN.POSITION IN (1) AND
  NOT EXISTS (SELECT *
   FROM TOKEN JOIN PROCESSES AS ENABLED  ON TOKEN.PROCESS_ID=ENABLED.ID
   WHERE TOKEN.POSITION=2 AND INHIBITED.ID=ENABLED.ID
 ))
```

For instance, for the first (upper) inbound connector, the query returns those process instances where there is a token in one of the upstream connectors' positions 4, 3, 1 and where there is no other token directly on the inbound connector (labeled 5). Finally the following query ties together the `CANDIDATES` and `INHIBITORS` views to yield the complete triggering condition of the OR join gateway:

```
SELECT
 "..." AS artifactId, "OR JOIN" AS artifactType, c.processId AS processId,
 c.tokenId1 AS tokenId, "tokenId2=" || c.tokenId2 AS customParameters
FROM CANDIDATES AS c, PROCESSES AS p
WHERE p.LIFECYCLE_PHASE = "running" AND p.ID = c.processId AND
 c.processId NOT IN (SELECT processId FROM INHIBITORS)
```

The associated stored procedure (representing the OR join rule's "action" part), needs to place a token onto the gateway's outbound connector and remove the candidate tokens from its inbound connectors:

```
CREATE PROCEDURE MYPROCESS.<artifactId> (
 IN processId INTEGER, IN tokenId INTEGER,
 IN parameters BPM.PARAMETER_TYPE)
LANGUAGE SQLSCRIPT AS
 tokenId2 INTEGER;
BEGIN
 SELECT VALUE INTO tokenId2 FROM :parameters WHERE NAME='tokenId2';
 INSERT INTO TOKENS(ID, PROCESS_ID, POSITION)
  VALUES(<new unique id>, :processId, <outbound position>);
 DELETE FROM TOKENS WHERE ID IN (:tokenId, :tokenId2);
END;
```

Depending on the number of candidate tokens, tokenId or tokenId2 may be NULL (i.e., there is no candidate token on the first or second inbound connector, respectively). The primary key of the to-be-generated outbound token can be retrieved from a sequence or a UID generator function of the database. The

gateway's outbound connector label ⟨*outbound position*⟩ is hard-coded into the procedure at compile time.

4.2.3. Assembly Stage

Finally, the queries and stored procedures that were generated for each process artifact need to be assembled into a single deployable SQL file, which represents the compiled process model. The principle is to aggregate the queries into a single view `QUEUE`, which represents the ready-to-execute process artifacts at any point in time:

```
CREATE VIEW MYPROCESS.QUEUE(artifactId, artifactType,
 processId, tokenId, customParameters) AS
 -- sub-query for process step 1
 UNION
 -- sub-query for process step 2
 UNION
 ...
```

The `QUEUE` view is placed into a process-specific schema `MYPROCESS`, such that the database definitions of different process models reside in different (named) schemas. A `SCHEDULER` procedure recurrently queries the `QUEUE` view until an empty result set is returned, signaling the end of the synchronous process fragment:

```
CREATE PROCEDURE MYPROCESS.SCHEDULER ()
LANGUAGE SQLSCRIPT AS
 processId INTEGER; tokenId INTEGER;
 hasTransition INTEGER; artifactId VARCHAR(100);
 artifactType VARCHAR(100); customParameters VARCHAR(1000);
BEGIN
 queue = SELECT * FROM MYPROCESS.QUEUE;
 CALL BPM.NEXT_TRANSITION(:queue, hasTransition, artifactId, artifactType,
  processId, tokenId, customParameters);
 WHILE (hasTransition > 0) DO
  CALL BPM.DECODE_PARAMETERS(customParameters, parameters);
  IF (artifactId = '<artifactId1>') THEN
   CALL MYPROCESS.<storedProcedure1>(processId, tokenId, :parameters);
  ELSEIF (artifactId = '<artifactId2>') THEN
   CALL MYPROCESS.<storedProcedure2>(processId, tokenId, :parameters);
  ...
  END IF;
  queue = SELECT * FROM MYPROCESS.QUEUE;
  CALL BPM.NEXT_TRANSITION(:queue, hasTransition, artifactId, artifactType,
   processId, tokenId, customParameters);
 END WHILE;
END;
```

The SCHEDULER procedure invokes another procedure "NEXT_TRANSITION" which is part of the BPM runtime schema. It provides a mechanism to look up a ready-to-run artifact by querying the QUEUE view (omitted for brevity).

As long as NEXT_TRANSITION yields a value of > 0 for the hasTransition output parameter, the SCHEDULER procedure will call the corresponding stored procedure for the queried ready-to-run artifact. Each synchronous process segment starts with an asynchronous artifact, i.e., depends on an outside event to be "triggered". We discuss the mechanics of invoking a synchronous process in the following section.

4.3. Asynchronous Artifacts Mapping

Executing a process model containing asynchronous artifacts requires spreading a single process instance across multiple database transactions, where a each transaction runs a single "synchronous segment". A synchronous segment is a process (fragment) having an asynchronous artifact as its starting point and purely synchronous segments downstream. Our approach to supporting processes interspersed with asynchronous artifacts entails (1) **synchronous segment decomposition** (identifying the set of synchronous segments which collectively make up the end-to-end process model), (2) **synchronous segment mapping** (mapping each synchronous segment into a set of SQL artifacts and stored procedures, following the instructions given in Section 4.2), and (3) **trigger definitions** (identifying the asynchronous starting point of each synchronous segment and mapping it onto at least one database trigger definition). We subsequently sketch out how we tackle the synchronous segment decomposition and trigger definitions tasks and give the corresponding algorithms in pseudo-code and by means of examples.

4.3.1. Synchronous Segment Decomposition

Part of our transformation of BPMN-based process models into the HANA programming model is to decompose the process graph into "synchronous segments". Informally, a synchronous segment is a contiguous fragment of the

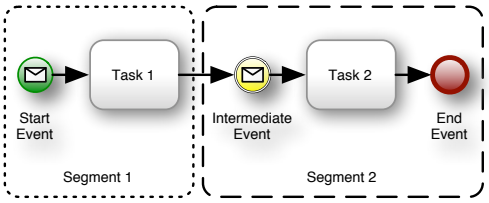

Figure 5: Simple synchronous segment decomposition

process model where the contained process artifacts can be executed in a single database transaction. As a consequence, a synchronous segment must only contain non-blocking process steps and after the synchronous segment was started

in some transaction T_1 it must not depend on some other transaction T_2 to successfully commit. Any (asynchronous) artifact in a process model that establishes a dependency on some other (external) transaction denotes the boundary of a synchronous segment. The previous synchronous segment(s) end(s) before that asynchronous artifact and a new synchronous segment starts having that artifact as its entry point. The idea is to later de-compose the process model such that it is completely covered by synchronous segments. Each synchronous segment is executed in the transaction that triggers its first (asynchronous) artifact. That is, if a process blocks on some asynchronous artifact, it is continued *within* the external transaction that "delivers" the event which resumes the process instance.

Figure 5 illustrates the synchronous segment decomposition of a simple process model, which is a plain sequential arrangement of artifacts. The process contains two asynchronous artifacts, being a *Start Event* and an *Intermediate Catch Event*. The remaining artifacts (two *Tasks* and an *End Event*) are synchronous. Accordingly, the first synchronous segment (surrounded by a dotted line) has the *Start Event* a its asynchronous starting point and the second synchronous segment (surrounded by a dashed line) has the *Intermediate Catch Event* as its starting point.

Sync-Async Artifacts. Before a process model can be decomposed into its synchronous segments, all sync-async artifacts need to be expanded into purely synchronous and asynchronous artifacts. For an illustration, please refer to Figure 6, where a *User Task* artifact is broken up into a sequence of a (synchronous) *Service Task* ("Create Task") and an (asynchronous) *Intermediate Catch Event* ("Wait for Task Completion Message"). Accordingly, the first synchronous seg-

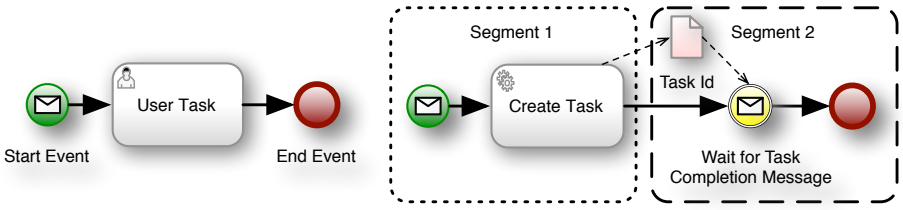

Figure 6: Decomposition of sync-async artifacts

ment (dotted border) encompasses the "Create Task" *Service Task*, whereas the "Wait for Task Completion Message" *Intermediate Catch Event* becomes the asynchronous starting point of the second synchronous segment (dashed border). The "Create Task" *Service Task* is responsible for creating the task instance, e.g., by synchronously invoking an external task management software. The "Wait for Task Completion Message" *Intermediate Catch Event* blocks the process until it receives a message from the task management software, signaling the completion of the respective task instance. A "Task ID" *Data Object*

provides for a unique identifier of the task instance which is created in the "Create Task" *Service Task*. The downstream "Wait for Task Completion Message" *Intermediate Catch Event* uses the value stored in the "Task ID" *Data Object* to match the completion message to the correct task instance.

In a more complex case, a *User Task* may require treatment of tasks exceeding some processing timeout. In BPMN, this could be handily modeled with *Boundary Catch Events*, which provide an extra "channel" for events out of running activities. For an illustration, please refer to Figure 7, where the upper part shows a process with a *User Task*, having a Timer Boundary Event attached, leading to an alternate branch ("Task 3"). The lower part of Figure 7

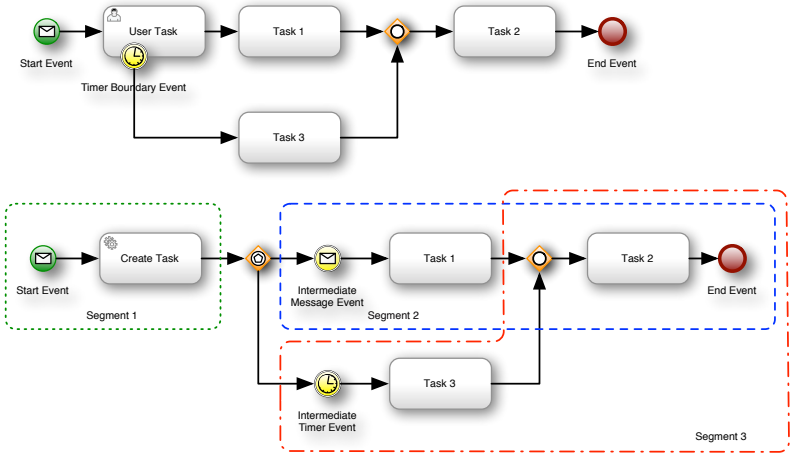

Figure 7: Complex decomposition in the presence of sync-async artifacts

shows the representation of that process after eliminating the sync-async *User Task* artifact with the corresponding boundaries of the resulting synchronous segments. The *User Task* was split into an upstream *Service Task* ("Create Task") and a downstream *Event-based Gateway* having two (mutually exclusive) *Intermediate Catch Events* connected to it. The first *Intermediate Catch Event* has a message trigger and represents the regular completion of the task. The second *Intermediate Catch Event* has a timer trigger and represents the timeout of the task before it was completed by an end user. The two *Intermediate Catch Events* form the starting points of two distinct, yet overlapping synchronous segments.

Overlapping Segments. In many cases, synchronous segments exclusively group process artifacts with no overlap. As can be seen in Figure 7, distinct synchronous segments do sometimes overlap, though. Generally, overlapping synchronous segments may exist because of converging gateways merging two or more "branches", where each branch belongs to a different synchronous segment. Two control flow branches belong to different synchronous segments when their

asynchronous starting points are different. This is because any synchronous segment has only a single asynchronous starting point. Figure 8, depicts a sim-

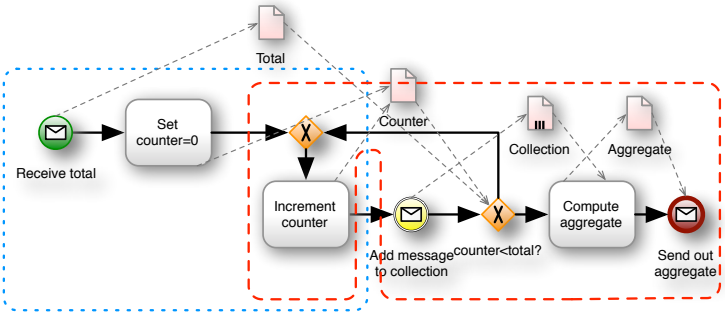

Figure 8: Overlapping synchronous segments

ple "message collect" pattern. A process instance receives a fixed number of messages and finally computes an aggregate value from the received messages. For instance, the process could be employed to sum up the prices of line items on some invoice. The first synchronous segment (dotted blue line) is anchored at the *Start Event*, where the process is initialized with the number of messages to receive, which is stored in a *Data Object* "Total". The first synchronous segment further comprises a downstream *Task* where another "Counter" *Data Object* is initialized to 0, a converging *Exclusive Gateway*, which demarcates the "bottom" of the loop where the messages are collected, and an "Increment Counter" *Task*, where the "Counter" *Data Object* is incremented by one.

The second synchronous segment (dashed red line) starts with the "Add message to collection" *Intermediate Catch Event* where the messages are received and appended to a list-valued "Collection" *Data Object*. A diverging "counter<total?" *Exclusive Gateway* then decides whether to stay within or to break out of the loop. The second synchronous segment incorporates both cases. In the former case, tokens are passed back to the upstream converging *Exclusive Gateway*, starting another loop cycle. In the latter case, a "Compute aggregate" *Task* is activated, which calculates a total value from the set of received messages that are stored in the "Collection" *Data Object*. Finally, an *End Throw Event* ends the process sends the aggregate value (stored in the "Aggregate" *Data Object*) to some external receiver.

The two synchronous segments overlap in portions of the loop where messages are progressively received. This is because the first synchronous segment enters the loop for the first time and only ends before the (asynchronous) *Intermediate Catch Event*, where the process blocks until a matching message was received. The second synchronous segment has that *Intermediate Catch Event* as its starting point. Each run of the second synchronous segment represents an iterations of that loop, which will repeatedly re-execute the initial portion of the loop cycle, thus causing the overlap with the first synchronous segment.

Decomposition Algorithm. The synchronous segment decomposition algorithm takes a process model as input and initially replaces the sync-async artifacts with their alternate representation, exclusively comprising purely synchronous and purely asynchronous artifacts (see Section 4.1). Starting at the artifacts that have no control flow predecessors, the algorithm then traverses the process model downstream, following control flow connectors. A synchronous segment S is defined as a tuple $S = \{a, T\}$, where a is the asynchronous starting point and T is the synchronous "tail" (i.e., the set of synchronous artifacts contained in S). Whenever an asynchronous artifact f is found, a new synchronous segment $(f, \{\})$ is created, having the newly found asynchronous artifact set as its asynchronous starting point. The algorithm can be written down in pseudo-code as follows:

```
procedure decomposeProcess
 in: Process p
 out: set of all synchronous segments A
begin
 Let A be an empty set {}.
 Replace sync-async artifacts.
 let F be the set of Flow Nodes contained in p, which have no incoming connectors.
 for each f in F do
  Assert that f is an asynchronous artifact.
  Call traverseSegment(f, (f, {}), A).
 end for
end
```

The algorithm `decomposeProcess` is invoked on an entire BPMN *Process* entity and invokes another algorithm `traverseSegment` for each of its *Flow Nodes* that have no incoming connectors. The algorithm `traverseSegment` first checks if the given *Flow Node* is an "new" asynchronous artifact (i.e., it is different from the current synchronous segment's starting point). If so, it will create a new synchronous segment, add it to the set of all discovered synchronous segments and recursively invoke `traverseSegment`. If the given *Flow Node* is placed within the current synchronous segment (i.e., it is either part of its synchronous "tail" or it is the segment's synchronous starting point), `traverseSegment` will make it part of the current synchronous segment and recursively invoke `traverseSegment` for any successor artifacts (i.e., *Flow Nodes* having an inbound *Sequence Flow* from the current artifact).

```
procedure traverseSegment
 in: FlowNode f
 inout: synchronous segment S,
  set of all synchronous segments A
begin
 if f is an asynchronous artifact and f is not identical to S.a then
  Let S be (f, {}).
  Include S in A.
```

```
  Call traverseSegment(f, (f, {}), A).
 else if f is not contained in S.A then
  if f is a synchronous artifact then
   Include f in S.T.
  end if
  Let N be the set of all successor FlowNodes of f.
  for each n in N do
   Call traverseSegment(n, S, A).
  end for
 end if
end
```

When `decomposeProcess` returns, the set A will contain all discovered synchronous segments.

4.4. Trigger Definitions

After having identified a process' synchronous segments, each synchronous segments' asynchronous starting point is mapped into one or many database trigger definitions, each representing the condition for that synchronous segment to be run. In this paper, we merely present an approach where each synchronous segment runs in the single transaction (which is shared with the database trigger). As a result, the number of operations performed in that transaction depends on the "length" of the synchronous segment (i.e., the number of BPMN artifacts within the synchronous segment).

Having few transactions per process limits the per-transaction overhead, but may lead to resource contention in concurrent transactions. For a number of reasons, we argue that this design is nevertheless well suited for in-memory databases. Firstly, the rows that are frequently updated by process steps affect tables with little potential for conflicting table row accesses. For example, the tokens are are passed through a synchronous segment (as the associated rows of the `TOKENS` table) are exclusive to that synchronous segment. Likewise, business objects from external applications that may be updated by a synchronous segment are typically bound to a single "case" (e.g., a purchase order business object is primarily accessed within the associated purchase process). Secondly, the database operations performed by the process steps within a synchronous segment are non-blocking and hard-wired (static per process model), following the characteristics of most real-world OLTP applications, thus generally well-suited for NewSQL databases [25]. Thirdly, most real-world processes that we have come across relatively frequently intersperse synchronous artifacts with asynchronous process steps (such as *User Tasks*, *Intermediate Catch Events*, etc.). And finally, our approach trades the number of database trigger definitions that are needed to represent a process model against the number of operations within each single transaction. With a 1-2 orders of magnitude speed-up of OLTP operations in an in-memory database ("scale-up") we believe that a reasonably large number of operations per transaction is within the "design parameters" of an in-memory database.

We subsequently illustrate the definitions of database triggers for (message-triggered) *Start Events* and *Intermediate Catch Events*.

Start Events. A process' start event initially kicks off a process instance by constructing the records for the `TOKENS` and `PROCESSES` tables. A *Start Message Event* specifically needs to receive a "message" to launch the process. In a traditional integration middleware environment, a message normally relates to a piece of data that is received from some external sender, such as an XML or JSON document which is extracted from the HTTP payload of some incoming Web request. In an in-memory database setting, incoming messages could be represented as records being inserted into a `MESSAGES` table. The *Start Message Event's* trigger could thus relate to insertion operations performed on said `MESSAGES` table:

```
CREATE TRIGGER MYPROCESS_SE_<artifact id>_MESSAGE_TRIGGERED
 AFTER INSERT ON MESSAGES FOR EACH ROW
 EXECUTE PROCEDURE MY_PROCESS.SCHEDULER()
```

Whenever a record is inserted into the `MESSAGES` table, it invokes the `SCHEDULER` procedure from the schema that was generated for the associated synchronous segment (c.f., Section 4.2.3).

Additionally, the *Start Event* needs to have a representation as an ECA rule, i.e., a database query constituting the condition part and a stored procedure for the action part:

```
SELECT
 "..." AS artifactId, "START MESSAGE EVENT" AS artifactType
 NULL AS processId, NULL AS tokenId,
 "messageId=" || m.ID AS customParameters
FROM MESSAGES AS m WHERE m.ID=NEW.ID AND <start condition>
```

The query's purpose was to test a *Start Event's* (optional) start condition, which is a Boolean predicate atop the message "payload". This predicate is to be placed in the "where" clause of the query and can access the attributes of the newly inserted `MESSAGES` record by means of the `NEW` prefix. Notice that both the `processId` and `tokenId` attributes are set to `NULL`. This is because neither the process instance nor the process' initial token exist before the start event is executed.

```
CREATE PROCEDURE MYPROCESS.<artifactId> (
 IN processId INTEGER, IN tokenId INTEGER,
 IN parameters BPM.PARAMETER_TYPE)
LANGUAGE SQLSCRIPT AS
 newProcessId INTEGER; newTokenId INTEGER;
BEGIN
 // generate unique newProcessId
 // and newTokenId values
 INSERT INTO PROCESSES(ID, LIFECYCLE_PHASE) VALUES(:newProcessId, ''running'');
```

```
 INSERT INTO TOKENS(ID, PROCESS_ID, POSITION) VALUES(:newTokenId, :newProcessId,
   <outbound connector label>);
 // perform output data mapping from message
 DELETE FROM MESSAGES WHERE ID IN
  SELECT VALUE FROM :parameters WHERE NAME=‘messageId’;
END;
```

The corresponding stored procedure initially generates unique primary keys for the newly inserted `TOKENS` and `PROCESSES` records (details omitted for brevity). After inserting these records into the respective tables, the stored procedure for *Start Events* may (optionally) perform a data mapping from the “received” message to a private process context (i.e., the set of its modeled *Data Objects*). Finally, the message that has triggered the *Start Event* is being deleted.

Intermediate Catch Events. Message-triggered *Intermediate Catch Events* expand the semantics of *Start Events* to a reliable, correlation-based message receipt [20] within running process instances. Generally, *Intermediate Catch Events* are used to synchronize a process to external events, being database state changes originating from other applications (or processes). By being embedded into the control flow of running process instances, *Intermediate Catch Events* may be triggered by two stimuli: a matching message arriving and a token reaching the *Intermediate Event* artifact. Both may independently trigger the *Intermediate Catch Event* as shown in the trigger definitions below:

```
CREATE TRIGGER MYPROCESS_IE_<artifact id>_TOKEN_TRIGGERED
 AFTER INSERT ON MESSAGES FOR EACH ROW
 EXECUTE PROCEDURE MYPROCESS.SCHEDULER();

CREATE TRIGGER MYPROCESS_IE_<artifact id>_TOKEN_TRIGGERED
 AFTER INSERT OR UPDATE OF POSITION ON BPM.TOKENS FOR EACH ROW
 EXECUTE PROCEDURE MYPROCESS.SCHEDULER();
```

The two database trigger definitions relate to new `MESSAGES` records being inserted and `TOKEN` records being inserted (or the `POSITION` attribute of existing `TOKEN` records being updated).

```
SELECT
 "..." AS artifactId, "INTERMEDIATE MESSAGE EVENT" AS artifactType
 PROCESSES.ID AS processId, TOKENS.ID AS tokenId,
 ‘messageId=’ || m.ID  AS customParameters
FROM
 MESSAGES AS m, PROCESSES AS p, TOKENS AS t
WHERE
 t.PROCESS_ID=p.ID AND m.ID=NEW.ID AND t.POSITION=<inbound connector label>
 AND <correlation condition>
```

Unlike *Start Events*, *Intermediate Catch Events* need to consider a token belonging to a running process instance. The “where” clause also incorporates

a check whether the token has actually reached the *Intermediate Catch Event* and a custom "correlation condition" holds, which is a test whether or not a message is being "matched" by the process instance. The corresponding stored procedure simply moves the token to behind the *Intermediate Catch Event* (by changing its `POSITION` attribute accordingly), (optionally) performs an output mapping from the message's "payload" to the private process context and finally removes the message record from the database:

```
CREATE PROCEDURE MYPROCESS.<artifactId> (
 IN processId INTEGER, IN tokenId INTEGER,
 IN parameters BPM.PARAMETER_TYPE)
LANGUAGE SQLSCRIPT AS
BEGIN
 UPDATE TOKENS
  SET POSITION=<outbound connecto label> WHERE ID=:tokenId;
 // perform output data mapping from message
 DELETE FROM MESSAGES WHERE ID IN
  SELECT VALUE FROM :parameters WHERE NAME='messageId';
END;
```

5. Contribution and Future Work

In summary, in-memory computing represents an opportunity for business process management to reach into wholly new scenarios, substantially increasing the overall value proposition of BPM. At the same time, today's middleware-based BPM systems could be in part obsolete as applications consolidate on in-memory databases. Both the opportunity and the threat form the motivation for our work.

In this paper, we proposed a BPM automation solution that is embedded into an in-memory database. Embracing the aforementioned stack consolidation approach, our design eliminates the need for an external application integration middleware system. Instead, our BPM system comprises two parts: (1) a runtime system which is implemented by means of the in-memory databases programming model and (2) a compiler infrastructure which maps business process definitions (here: BPMN 2.0 models) into native artifacts of the in-memory database which interact with the aforementioned runtime system.

Both the runtime system and the compiled process models constitute artifacts of the in-memory database's programming model (e.g., SQL and stored procedures). By deploying these artifacts into the in-memory database, our architectural approach reuses fundamental capabilities of the in-memory database (such as transactional concurrency control; failover and persistence; cluster-readiness and scale-out; backup, recovery, and archiving; etc.) and abolishes the need for a separate BPM stack. This design stands in pronounced difference to an established BPM middleware architecture where the core BPM functionality (in terms of process orchestration, monitoring, administration, lifecycle management, etc.) are normally provided by a separate BPM runtime which

can be a designated middleware "stack" or be a component of a larger application server. For example, some ERP systems come with workflow management capabilities. These capabilities are provided by infrastructure components being part of the application server, which also hosts the core ERP components.

By running business processes inside of the in-memory database, our approach benefits from the following advances over state-of-the-art BPM systems:

1. Business processes running inside of an in-memory database benefit from the performance advances of in-memory databases as such. In effect, model-based business processes become a suitable way of implementing high-performance applications.
2. Most importantly, business processes that are based in the same in-memory database, which also hosts other applications have an unconstrained reach into the application data artifacts (such as business objects and events), thus being able to deeply integrate against these applications.
3. By reusing major capabilities of the in-memory database, the cost and technical risk of building a BPM system is much reduced. In fact, many non-functional characteristics can be relayed back to features of the in-memory database itself.

When have made our in-memory BPM automation approach available as open source[4] and invite interested readers to experiment with and contribute to this software. Our current emphasis is on validating the performance and scale-out characteristics of our approach, which we hope to publish soon. In addition, we expand the coverage of the BPMN 2.0 standard in an upcoming publication and address the state synchronization challenges of integrating applications with business processes on the persistence level.

References

[1] van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M.: Business process management: A survey. In: Business Process Management, *LNCS*, vol. 2678, pp. 1–12. Springer (2003)

[2] Agrawal, A., Amend, M., Das, M., Ford, M., Keller, C., Kloppmann, M., König, D., Leymann, F., Müller, R., Pfau, G., et al.: Web services human task (ws-humantask), version 1.0., june 2007. Online at: http://www.adobe.com/devnet/livecycle/pdfs/ws humantask spec. pdf

[3] Bae, J., Bae, H., Kang, S.H., Kim, Y.: Automatic control of workflow processes using eca rules. IEEE TKDE **16**(8), 1010–1023 (2004)

[4] Balko, S., Hettel, T.: Systems and methods providing a token synchronization gateway for a graph-based business process model (2013). US Patent 8,453,127

[5] Boncz, P.A., Kersten, M.L.: Monet. an impressionist sketch of an advanced database system. In: In Proc. IEEE BIWIT workshop. Citeseer (1994)

[4] https://github.com/sbalko/bpmn2hana

[6] Brewer, E.A.: Towards robust distributed systems (abstract). In: PODC, p. 7 (2000)

[7] Bry, F., Eckert, M., Pătrânjan, P.L., Romanenko, I.: Realizing business processes with ECA rules: Benefits, challenges, limits. Springer (2006)

[8] Cattell, R.: Scalable sql and nosql data stores. ACM SIGMOD Record **39**(4), 12–27 (2011)

[9] Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al.: Spanner: Googles globally distributed database. ACM Transactions on Computer Systems (TOCS) **31**(3), 8 (2013)

[10] DeBrabant, J., Pavlo, A., Tu, S., Stonebraker, M., Zdonik, S.: Anti-caching: A new approach to database management system architecture. Proceedings of the VLDB Endowment **6**(14), 1942–1953 (2013)

[11] Diaconu, C., Freedman, C., Ismert, E., Larson, P.Å., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: Sql server's memory-optimized oltp engine. In: SIGMOD, pp. 1243–1254 (2013)

[12] Eckerson, W.W.: Three tier client/server architecture: Achieving scalability, performance, and efficiency in client server applications. Open Information Systems **10**(1) (1995)

[13] Färber, F., Cha, S.K., Primsch, J., Bornhövd, C., Sigg, S., Lehner, W.: Sap hana database: data management for modern business applications. SIGMOD Record **40**(4), 45–51 (2011)

[14] Färber, F., May, N., Lehner, W., Große, P., Müller, I., Rauhe, H., Dees, J.: The sap hana database – an architecture overview. IEEE Data Eng. Bull. **35**(1), 28–33 (2012)

[15] Fowler, M.: Patterns of Enterprise Application Architecture, first edition edn. Addison-Wesley, Boston (2003)

[16] Gable, J.: Enterprise application integration: Eai is the soluble glue needed for modular relationships that allow organizations to be flexible and responsive to market demands. Information Management **36**(2) (2002)

[17] Garcia-Molina, H., Salem, K.: Main memory database systems: An overview. IEEE Trans. Knowl. Data Eng. **4**(6), 509–516 (1992)

[18] Han, J., Haihong, E., Le, G., Du, J.: Survey on nosql database. In: Pervasive computing and applications (ICPCA), 2011 6th international conference on, pp. 363–366. IEEE (2011)

[19] Leavitt, N.: Will nosql databases live up to their promise? IEEE Computer **43**(2), 12–14 (2010)

[20] Model, B.P.: Notation (bpmn), v. 2.0, 2011. OMG: www. omg. org/spec/BPMN/2.0

[21] Raman, V., Attaluri, G.K., Barber, R., Chainani, N., Kalmuk, D., KulandaiSamy, V., Leenstra, J., Lightstone, S., Liu, S., Lohman, G.M., Malkemus, T., Müller, R., Pandis, I., Schiefer, B., Sharpe, D., Sidle, R., Storm, A.J., Zhang, L.: Db2 with blu acceleration: So much more than just a column store. PVLDB **6**(11), 1080–1091 (2013)

[22] Reese, W.: Nginx: the high-performance web server and reverse proxy. Linux Journal **2008**(173), 2 (2008)

[23] Stonebraker, M.: Sql databases v. nosql databases. CACM **53**(4), 10–11 (2010)

[24] Stonebraker, M., Cattell, R.: 10 rules for scalable performance in'simple operation'datastores. Communications of the ACM **54**(6), 72–80 (2011)

[25] Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era:(it's time for a complete rewrite). In: Proceedings of the 33rd international conference on Very large data bases, pp. 1150–1160. VLDB Endowment (2007)

[26] Stonebraker, M., Pavlo, A., Taft, R., Brodie, M.L.: Enterprise database applications and the cloud: A difficult road ahead. In: 2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, March 11-14, 2014, pp. 1–6 (2014)

[27] Tilkov, S., Vinoski, S.: Node. js: Using javascript to build high-performance network programs. IEEE Internet Computing **14**(6), 0080–83 (2010)

[28] Vogels, W.: Eventually consistent. CACM **52**(1), 40–44 (2009)

[29] Völzer, H.: A new semantics for the inclusive converging gateway in safe processes. In: Business Process Management, pp. 294–309. Springer (2010)

[30] Wynn, M.T., Edmond, D., van der Aalst, W.M., ter Hofstede, A.H.: Achieving a general, formal and decidable approach to the or-join in workflow using reset nets. In: Applications and Theory of Petri Nets 2005, pp. 423–443. Springer (2005)