Available-To-Promise on an In-Memory Column Store

Christian Tinnefeld[1], Stephan Müller[1], Helen Kaltegärtner[1], Sebastian Hillig[1], Lars Butzmann[1], David Eickhoff[1], Stefan Klauck[1], Daniel Taschik[1], Björn Wagner[1], Oliver Xylander[1], Cafer Tosun[2], Alexander Zeier[1], and Hasso Plattner[1]

[1] Hasso Plattner Institute, University of Potsdam, August-Bebel-Str. 88, 14482 Potsdam, Germany, Email: (firstname.lastname)@hpi.uni-potsdam.de

[2] SAP AG, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany, Email: (firstname.lastname)@sap.com

Abstract: Available-To-Promise (ATP) is an application in the context of Supply Chain Management (SCM) systems and provides a checking mechanism that calculates if the desired products of a customer order can be delivered on the requested date. Modern SCM systems store relevant data records as aggregated numbers which implies the disadvantages of maintaining redundant data as well as inflexibility in querying the data. Our approach omits aggregates by storing all individual data records in an in-memory, column-store and scans through all relevant records on-the-fly for each check. We contribute by describing the novel data organization and a locking-free, highly-concurrent ATP checking algorithm. Additionally, we explain how new business functionality such as instant rescheduling of orders can be realized with our approach. All concepts are implemented within a prototype and benchmarked by using an anonymized SCM dataset of a Fortune 500 consumer products company. The paper closes with a discussion of the results and gives an outlook how this approach can help companies to find the right balance between low inventory costs and high order fulfillment rates.

1 Introduction

There is an ongoing discussion in the database community to what extent applications can benefit from a database management system (DBMS) that exactly suits their needs. One central paper in this discussion is written by Stonebraker and Cetintemel [Sc05] who argument that applications such as text search, scientific applications, data warehousing, and stream processing can benefit from a performance, maintenance, and functionality perspective by using application specific database engines. As stated by Krueger et al. [KTG$^+$10], we think that this statement is also true for the domain of traditional enterprise applications which we want to exemplify in this paper with the ATP application.

ATP provides a checking mechanism to obtain feasible due dates for a customer order. This is done by comparing the quantities of products which are in stock or scheduled for production against the quantities of products which are assigned to already promised orders [Dic05, SZ03]. A common technique in current SCM systems is using aggregated values for keeping track of the different quantities, which results in having a separate aggregate for each different product. This means that e.g. a new product in stock would increase the value of such an aggregate while the assignment of products to a confirmed customer order would decrease it. Although the use of aggregates reduces the necessary amounts of I/O operations and CPU cycles for the single ATP check itself, it introduces the following disadvantages:

Redundant Data. One problem that arises in association with materialized aggregates is the need for data replication and therefore for complex synchronization strategies [Pla09]. In order to preserve a consistent view on the data across the whole system, every write operation has to be propagated to all replications. Even if the updates are triggered immediately, they still imply delays causing temporary inconsistencies. Additionally, even if the amount of I/O operations and CPU cycles is reduced to a minimum for the check itself by using aggregates, the overall sum of needed operations might be higher due to synchronization as well as maintenance and costly back calculation of the aggregates.

Exclusive Locking. A related issue consists of locking for update operations. All modifications to an aggregate require exclusive access to the respective database entity and block concurrent read and write processes. The downside of locking is obvious, as it queues the incoming requests and affects the performance significantly in case of a highly parallel workload.

Inflexible Data Querying. Evidently, the gain in performance concerning isolated queries comes at the cost of less flexibility. For ATP systems in particular, this fixedness poses major restrictions. The rolled up data structures are tailored for a predefined set of queries. Unforeseeable operations referring to attributes that were not considered at design time cannot be answered with these pre-aggregated quantities. Those attributes include for instance shelf life, product quality, customer performance and other random characteristics of products, orders, or customers. Additionally, due to the use of aggregates the temporal granularity of the check is fixed. Once the aggregates are defined and created based on e.g. the available quantities per day, it is not possible to perform ATP checks on an hourly granularity.

Inflexible Data Schema Extensions. The previously mentioned inflexibility of not being able to change the temporal granularity of a single check indicates another related disadvantage: the inability to change the data schema once an initial definition has been done. The change of the temporal check granularity or the inclusion of a previously unconsidered attribute is only possible with a cumbersome reorganization of the existing data.

No Data History. Maintaining aggregates instead of recording all transactions enriched with information of interest means to lose track of how the aggregates have been modified. In other words, no history information is available for analytics or for rescheduling processes.

As stated above, the evolution of business needs indicates an increasing relevance of sophisticated analytical applications. In order to realize immediate answer to arbitrary analytical queries without long lasting ETL processes, raw data in a format that enables on-the-fly processing is needed. Facing these demands, recent trends are heading towards column-oriented in-memory databases. In-memory databases keep all data in main memory and therefore facilitate fast and random access. To be able to hold billions of data records in memory, high compression rates are mandatory. By storing the data column-wise, generally suitable compression techniques can be applied. The main reason for good compression results achievable in column-stores is the similarity of the entries in a column, since the compression ratio is often dominated by the number of distinct values [KGT$^+$09].

Another advantage coming along with column-stores is that many operations, notably aggregations, equi-joins, and selections, can be performed directly on compressed data [AMH08, GS91]. Consequently, fewer entries have to be decompressed for reconstruction of tuples. This strategy called lazy decompression [CGK01] helps to save CPU cycles for decompression. Reducing computing time is particularly in in-memory databases highly relevant, because I/O costs are extremely low, so that CPU time influences the overall execution time significantly [HLAM06]. This technique is especially beneficial in the context of insert-only as updates do not directly require a decompression of already stored values, but result in appending new values for already existing tuples. Furthermore, the read-optimized columns can go along with a smaller, write-optimized so called delta store which is used for updates.

Consequently, in column-oriented in-memory databases aggregates can be calculated by processing the appropriate column without the need to read the entire table from disk or decompress it in advance. Thus, column-oriented database systems can operate on huge datasets efficiently and thereby comply with the requirements of an OLAP system very well [BMK99]. They also bring along a new level of flexibility, since they are not confined to predetermined materialized aggregates. Write operations are not limited by the read-optimized data structure as they are performed in the delta store.

The remainder of this paper is organized in the following way: Section 2 presents the involved concepts which are needed for executing an ATP check on a columnar database. This includes aspects such as data organization, the check algorithm itself, and how to deal with concurrency. Section 3 includes the description of a prototypical implementation of the concepts and corresponding benchmarks which were done on an anonymized SCM dataset of a Fortune 500 consumer products company. The prototypical implementation has been done in the context of a joint research project between the Hasso Plattner Institute and SAP. Section 4 discusses the possible business implications of this approach by listing new or improved functionalities in the context of ATP. Section 5 concludes with a discussion of the results and an outlook how this approach could be used for taking ATP to the next level by providing a profit-maximizing ATP checking mechanism.

2 Involved Concepts

With a general overview of the ATP check and its limitations with the use of aggregates, this section provides the underlying concepts of the prototypical ATP application based

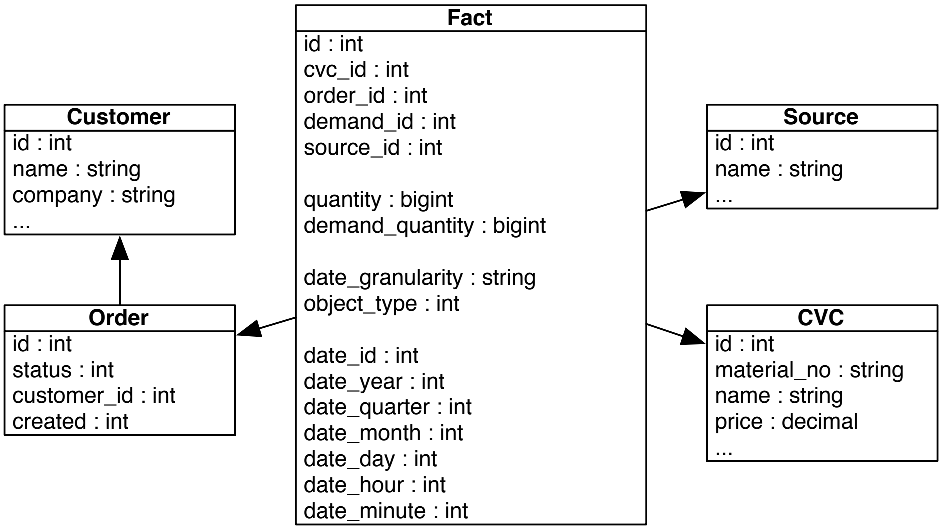


Figure 1: Simplified physical schema as used in prototype

on a columnar, in-memory database. Subsection 2.1 introduces the main classes in the ATP data model and their physical representation in the database. The subject of Subsection 2.2 are two algorithms to calculate the due dates focusing on their applicability on a column-oriented database system. In Subsection 2.3 finally the strengths and weaknesses of different strategies to handle concurrent ATP requests are discussed.

2.1 Data Organization

In this subsection, a simplified data model sufficient for a basic understanding of the prototype is presented. The information relevant to an ATP system are primarily line items of sales orders also referred to as customer demands, delivery promises or conducted outputs, the stock level, and planned production inputs. These types of transaction data are consolidated in one table with an object_type column for identification, forming the fact table in the star schema [Mar04].

Essentially, an entry in the fact table indicates how many items of a product leave or enter the stock on a specified date. The products are listed by their characteristic value combinations (CVC), to be uniquely identified. In the fact table, there are two quantity columns. This is due to the fact that the customer demands indicate what was ordered and do not represent planned stock movements, which are stored in the promises. To be able to provide information about the inventory, only planned and conducted stock movements but not the customer demands have to be added up. For this reason, the customer demands have a separated quantity column, the demand_quantity. This way, they do not have to be filtered when aggregating the stock movements in the quantity column. Recurring data,

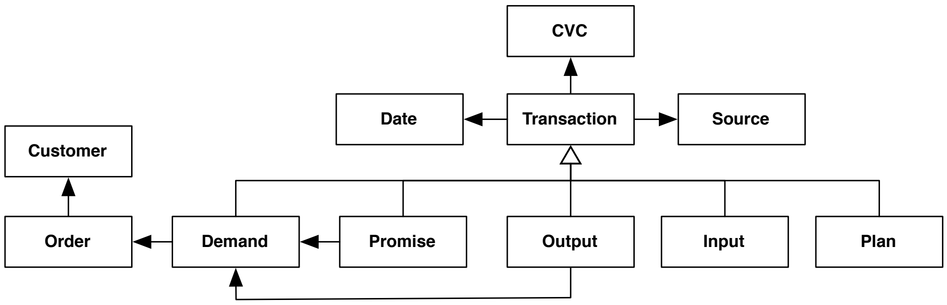


Figure 2: Simplified ERM of the ATP process

id	date_id	cvc_id	demand_id	demand_quantity	quantity	object_type
5	1286668800	1	5	-45	0	3
6	1286668800	1	5	0	-45	1
7	1286409600	1	0	500	500	2
8	1286323200	2	3	-10	0	4

Table 1: Fact table extract

such as the customer who ordered the product or the source of an input, be it a production plant or supplier is stored in dimension tables.

To avoid expensive joins with a potentially large Date table, this dimension is de-normalized, accepting a certain degree of redundancy. The physical data model as implemented in the prototype is shown in Figure 1. For the sake of clarity, the following examinations abstract from this and other optimizations. The business objects used in the application are illustrated in Figure 2.

As a typical feature of a star schema, most columns in the fact table are foreign keys to dimension tables. It can be expected that the content of the majority of dimension tables is static. The entries in the Date table will only be updated, when the fiscal year comes to an end and the planning horizon changes. Just as reasonable is the assumption that the content of the CVC and Source tables is constant. Changes to these tables are only necessary, if the company introduces a new product, puts a production plant into operation, or goods are purchased from a new supplier. As a consequence, the total volume of data can be reduced by storing recurring information in dimension tables.

In Table 1 an extract from the fact table, with the redundant date specifications date_year to date_minute as well the columns order_id, source_id, and date_granularity left out, is provided. The first row with id 5 represents a customer demand with the computed delivery promise in the second row. This relation can be seen by the foreign key demand_id, which is set to 5 in the promise entry and hence points to the demand row. Besides, the object types 3 and 1 identify these rows as demand and promise. Since they correspond in quantity and date, the request can be fulfilled in time.

The third column is an input, characterized by the object_type 2, of 500 items to the same product that was requested by the afore mentioned demand. Inputs do not refer directly to a customer demand in order to stay as flexible as possible when it comes to unexpected loss of stock and a redistribution of resources is required. So, the foreign key demand_id is a null pointer. One might wonder, why the quantity is replicated in the column demand_quantity. This way, we retain the option to run the ATP check against the requested instead of the promised quantities and thereby favor already booked orders over new ones.

Object_type 4 identifies withdrawals from stock made to accomplish a customer order. Basically, a promise is turned into such an output, as soon as the items are taken from the warehouse and on their way to the customer. To save the connection between demand and outputs, the outputs also store the id of the demand in the dedicated foreign key column. The last row in Table 1 is an example for an output. The demand it refers to, the fact with id 4, is not listed in the extract.

2.2 ATP check Algorithms

The core functionality of an ATP application is the determination of earliest possible delivery dates based on the inventory. The available capacities accrue directly from stock, inputs, and previously confirmed orders. This subsection suggests and evaluates two solutions to this problem, addressing the decisive ATP requirements and the characteristics of our prototype. The two algorithms are equal in terms of correctness and complexity but differ in performance depending on the underlying architecture and dataset.

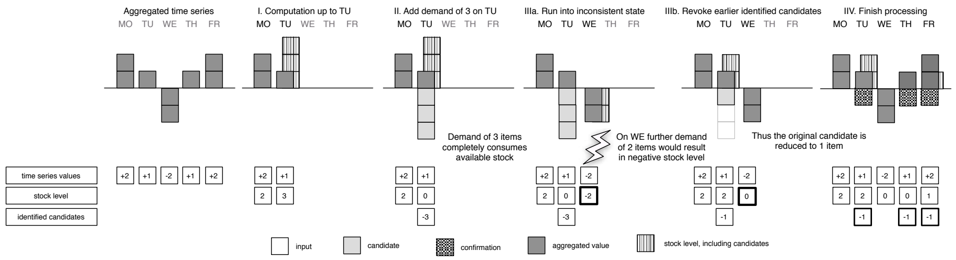
2.2.1 Candidate Check

The *candidate check* constitutes the first alternative to compute due dates. The name candidate check is derived from the supposed promises, the so called candidates, which are created in course of the check. Basically, the algorithm operates on a chronological list of dates associated with the aggregated inputs and outputs for the particular dates. Thereby, dates with a total quantity of zero will be omitted to reduce the runtime. Such (date, quantity) - tuples will be referred to as *buckets* and chronological lists of buckets as *time series*. Apart from the time series, the algorithm maintains a list of candidate buckets as temporary promises, also sorted by date in ascending order. As an interim promise, a candidate indicates the quantity that is allocated on the specified date for the current demand. The word temporary is meant to point out the potential deletion or reduction of the candidate while iterating over the remainder of the time series. Those candidates that can be retained until the algorithm terminates, will be written as promises to the database. If there is only one candidate with the desired quantity on the desired date, the order will be fulfilled in time.

In the following, details of the algorithm, particularly the determination of candidates, will be explained. To identify candidates, the total stock level for each date is required. Therefore, a new data structure is deduced from the original time series, henceforth termed

date	aggregated quantity	accumulated time series
MO	2	2
TU	1	3
WE	-2	1
TH	1	2
FR	2	4

Table 2: Accumulated time series



Aggregated time series
MO TU WE TH FR
I. Computation up to TU
II. Add demand of 3 on TU
IIIa. Run into inconsistent state
IIIb. Revoke earlier identified candidates
IIV. Finish processing
Demand of 3 items completely consumes available stock
On WE further demand of 2 items would result in negative stock level
Thus the original candidate is reduced to 1 item
time series values
stock level
identified candidates
+2 +1 -2 +1 +2
2 3
-3
2 0 -2
2 2 0
-1
2 2 0 0 1
-1 -1 -1
input
candidate
confirmation
aggregated value
stock level, including candidates

Figure 3: Candidate check example

accumulated time series as can be seen in Table 2.

From the very beginning of the planning horizon, all quantities are added up to the desired date. Reaching that point in the time series, the creation of candidates starts. If the accumulated quantity is positive, the first candidate will be initialized. Generally, the quantity of a candidate is limited to the stock level of the respective date, in this case the desired date. The maximum quantity of a candidate is logically the desired quantity. For further processing, the accumulated quantity is reduced by the quantity of the new candidate. As long as the desired quantity has not been completely allocated, candidates will be created while processing the time series.

When the stock level drops below zero due to other promises or a decline in production, the list of candidates has to be corrected. Thereby, the candidate quantities are deallocated until the stock level returns to zero or the list is exhausted. To ensure best possible delivery dates, the list is updated descendingly, removing the latest buckets first. The entire candidate algorithm is formally described in Listing 1. To highlight the essential control flow, the readjustment of the candidate list in case of a negative accumulated quantity is not listed on instruction level but hidden in the method *truncate_qty*, which is invoked on the candidate list, cf. Listing 1 Line 26.

To improve the understanding of the algorithm, a walk-through with a concrete example is undertaken. The available stock including one output, represented in the first diagram in Figure 3, and a new order of three items for Tuesday are the starting point for this excursion. The first diagram shows the time series with a negative bucket on Wednesday. This might be confusing in the first place, as it appears to be an overbooking. In fact, it is not an overbooking, which becomes obvious when calculating the accumulated time series as seen in Table 2. The three inputs on Monday and Tuesday compensate the outputs on Wednesday.

```
 1 def candidates(time_series, desired_date, desired_qty):
 2     candidates = []
 3     acquired = 0
 4     accumulated = 0
 5     for date, qty in time_series:
 6         accumulated += qty
 7         # do not start before desired_date
 8         if date < desired_date:
 9             continue
10         if accumulated > 0:
11             wanted = desired - acquired # pending quantity
12             if wanted > 0:
13                 if accumulated >= wanted:
14                     # total covering of wanted quantity
15                     candidates.append((date, wanted))
16                     acquired = desired
17                     accumulated -= wanted
18                 else:
19                     # partial covering
20                     candidates.append((date, accumulated))
21                     acquired += accumulated
22                     accumulated = 0
23         elif (accumulated < 0) and (len(candidates) > 0):
24             # acquired too much, give back until
25             # accumulated is not negative
26             truncate_qty(candidates, accumulated, acquired)
27     return candidates
```

Listing 1: Candidate Check Algorithm

Going one step back, the accumulated time series is built up to Tuesday. As already mentioned and reflected in Line 8 in Listing 1, the creation of candidates starts at the desired date, which is Tuesday in this example. The accumulated quantity, which results from the Monday and the Tuesday bucket, is three corresponding to the desired quantity. In compliance with Lines 15 to 17 in Listing 1, a candidate for Tuesday with three items is appended to the still empty candidate list, the already acquired quantity is set to three, and the accumulated quantity is cut to zero. This new candidate can be seen in diagram II in Figure 3. On the next day, two items leave the stock due to the output. Consequently, the accumulated quantity drops below zero as can be seen in IIIa. To compensate the overbooking, two items have to be given back. Therefore, the candidate list is processed backwards. Since there is only one entry holding three items, this candidate will be truncated to one, cf. diagram IIIb in Figure 3, and the acquired quantity will be reduced accordingly. Proceeding the same way, a candidate for Thursday with one item and one for Friday covering the last pending item will be appended to the candidate list. At the end of the planning horizon, on Friday, the accumulated quantity representing the stock level is still positive so that the three candidates will be returned as promises.

date	input	promises	net
MO	2	-1	1
TU	2	0	0
WE	1	-3	0
TH	3	-2	1
FR	1	0	1

Table 3: Net time series aggregates

With regard to the fact that per date either a candidate is created or existing candidates are removed, the candidate check features a linear complexity. Besides, one can easily see that this algorithm performs best in case of sufficient stock, because only one candidate is created and no updates are required.

2.2.2 Net Time Series Check

As stated above, there is a second approach, the *net time series check*, leading to the same promises and showing the same complexity. Since this algorithm was not integrated into the prototype for technological reasons, only the main idea will be outlined. The starting point for the net time series check are two time series, aggregating disjoint sets of fact table entries. One time series adds up the inputs and the other one all confirmed and conducted promises.

In a next step, bucket by bucket the aggregated promises are matched to the quantities of the input time series. A bucket from the promise time series consumes primarily items from the input bucket of the same date. If this corresponding input bucket does not satisfy the promise bucket, first earlier input buckets and after that later ones will be emptied. The resulting net time series represents the resources that are available at the time of the ATP request, cf. Table 3, and gives this algorithm its name. When the net time series is set up, the new demand acts just like the promise buckets and takes out the requested quantity from the net time series. Promises are created based on the obtained input buckets, .

Whereas the candidate check has to consider the whole planning horizon independent on the stock level, the net time series check can under certain conditions reach minimal computing times. If there are no promises or if they all fall into a small number of buckets and sufficient inventory is available, the algorithm only has to match a few buckets. Under the premise that they can even be fulfilled out of the preferred input buckets, only a fixed number of operations, which is determined by the number of promise buckets, is required and a constant complexity is reached. In such scenarios, the net time series check outperforms the candidate algorithm.This scenario is unlikely in production though, as it requires all confirmed promises to fall onto a few condensed dates.

2.2.3 Comparison

To sum up, the two presented algorithms deliver optimal promises to the customer and both vary in performance depending on the characteristics of the dataset. So, from a merely logical point of view both alternatives are equal. However, taking technological aspects into account, major differences can be identified.

```
    SELECT SUM(Fact.quantity), MAX(Fact.date_id)
    FROM Fact
    WHERE Fact.cvc_id = 1
    GROUP BY Fact.date_year, Fact.date_month, Fact.date_day
    ORDER BY MAX(Fact.date_id)
```

Listing 2: Candidate check aggregation

The descriptions above start with the initial time series already available. The creation of these data structures has not been treated so far. In fact, it is the most time consuming part in the overall ATP check, because the raw data has to be aggregated on-the-fly. To set up the time series for the candidate check, all inputs and promises of the specific product are added up grouped by the selected time granularity. The resulting database query for the granularity level day is shown in Listing 2. The maximum of all dates per bucket is selected, because depending on the granularity several timestamps belong into one aggregation class. It is necessary to find the latest timestamp of all aggregated records, to make sure that at this point in time all movements have already been issued.

For the net time series check, the situation is more complicated. Two separated time series are necessary, which can basically be achieved in two different ways. The first option would be to add an additional group-by attribute, the object-type respectively, to the query for the candidate check. The downside of this method lies in the structure of the result set, which includes the buckets for both time series. Thus, new empty time series are created and while iterating over the result set the buckets are inserted either into the input or the promise time series. Furthermore, with an increasing number of group-by attributes the query execution time increases substantially. Alternatively, the two time series can be extracted from the database in separate queries by adding another predicate for the object type. This approach obviously requires two full table scans and thus does not present a feasible solution, especially when the application has to deal with several millions of records.

Another disadvantage, which applies to both query variants equally, is the quantity of data to be transferred from the database to the application layer, as inputs and promises are not consolidated into one time series. Being aware of the disadvantages of the net time series check with respect to the database queries, the decision in favor of the candidate check becomes more transparent.

2.3 Concurrency Control

The algorithms listed in Subsection 2.2 focus on the execution of one ATP check in isolation. They do not factor in the difficulties caused by multiple parallel checks referring to the same product and consequently accessing the same data. The management of concurrent processes indeed belongs to the main challenges in parallel environments. Particularly in an ATP system, data correctness and consistency at any time is an essential requirement to avoid wrong promises.

To be precise, there is a temporal gap in between reading the current stock levels from the database and writing a promise in the end based on those results. In the meantime another process, proceeding from the same inventory, might have calculated a delivery date and booked resources that are necessary to fulfill the first request. As a consequence, the same products are promised twice causing an inconsistent state in the database. Such anomalies are a serious problem in an ATP system. A company might take severe damage from dealing with the resulting effects including angry customers, contractual penalties, costs for acquiring substitute products, and so on. In current ATP systems, the common practice is to serialize the execution of concurrent requests by locks. Our prototype allows for choosing out of three strategies suitable for different situation. These three approaches will be elaborated in this section, benchmarking results will be presented in Section 3.

2.3.1 Exclusive Lock

A naive but secure way to preserve consistency constraints is as mentioned above to lock critical data. In this context, it means to grant to one process exclusive access rights to the entire set of fact table entries for the desired product. Since always the whole planning horizon has to be checked, simultaneous checks on different dates are unrealizable. The first incoming process acquires the lock, queries the database, calculates the delivery dates, writes the promises back to the database, and finally releases the lock for the next process to start. Apparently, this locking policy, termed exclusive lock, involves superfluous latencies in case of sufficient stock. If there were several hundreds of incoming requests for one product per minute, a sequential schedule would lead to response times that exceed the limit of tolerance.

2.3.2 Optimistic

Whereas the exclusive lock queues incoming requests for the same product, the second solution, an optimistic strategy, enables parallel execution without blocking. Theoretically, the optimistic mechanism allows for as many parallel requests as cores available and therefore scales linearly with hardware resources. For now, it seems as if the gain in scalability implies a certain staleness of the data. Indeed, without modification of the ATP check, violations to the consistency may occur.

This modification consists of a consistency check after the original candidate check. A process has to verify the correctness of its result concerning the new stock level. The term optimistic expresses the nature of this strategy presuming a balanced stock situation. To avoid a second full table scan, the maximum row id of the fact table identifying the most recently written record is retrieved from the database in advance. Afterwards, the check is performed based on the stock level up to this id. It must be mentioned that the row id is a continuously incrementing counter, which facilitates absolute ordering of fact table entries by insertion time.

If the candidate check results in a complete or at least in a partial delivery, the promises are written to the database and a consistency check will be performed. For this purpose, all fact table entries with an id higher than the initially determined maximum id are retrieved. The result set contains exactly those inputs and promises that were recorded during the check. The records are successively included into the time series that comprises the entries up to the maximum id. In cases of outputs, for instance corrections to planned inputs or promises, the stock will be checked for overbooking and the invalid quantity will be saved. If the record is a promise related to the current demand, the system will use its quantity to compensate the overbooked quantity. So the promise will be either deleted or reduced. When all records are processed, a new ATP request will be triggered with the total rebooked quantity. Evidently, this conflict resolution procedure can end up in an infinite loop. In order to enforce a termination, one could define a maximum recursion depth and switch to the exclusive lock, once this depth is reached.

To sum up, the optimistic approach dispenses with locks, unless the inventory is close to exhausting and many parallel requests are competing for the last items. Though, as long as sufficient capacities are available, the avoidance of locks can be fully leveraged by providing appropriate hardware resources.

2.3.3 Instant Reservation

The third idea arises from the drawbacks of the two afore mentioned ones. In general terms, it works similarly to the optimistic strategy without the need for conflict resolution. The key to this enhancement lies in blocking the desired quantity before the check is started. So, this approach is called instant reservation. To reserve the requested quantity, a promise complying with the customer demand is written directly to the database. For the candidate check, only the fact entries up to this promise are aggregated so that the check will not be manipulated by its own reservation.

Once the result is computed, it is compared to the initially recorded promise. If they do not correspond to each other, an adjustment will follow. At this point, it must be mentioned that this process is totally transparent to the user of the system. The reservation promise will not be passed on to the user, unless it corresponds to the calculated one. Otherwise, the updated promises will be delivered.

Reviewing the sequence of steps, one might have noticed that the reservation can cause an overbooking. But through the comparison in the end, each process clears up self-inflicted inconsistencies. Concurrent processes include earlier reservations in their calculations. Accordingly, it is guaranteed that they do not allocate resources needed to fulfill other earlier processes. Even if the earlier processes have not finished their check yet, the desired quantities are blocked by the reservation.

However, in one specific case, this approach does not produce an optimal utilization of available resources. Given an empty stock and two incoming requests with the same quantity, the first one writes its reservation and initiates the aggregation excluding the reservation. Before the second request writes the reservation promise, an input is recorded with exactly the same quantity like the demands. This input is not seen by the first process so that the request will be rejected.

Therefore, it would be desirable that the input will be used to satisfy the second process. This process, however, includes both, the input and the reservation of the first process. Thus, it also faces an empty stock and refuses the customer request. Being aware of this problem, one has to decide, whether he will accept it and receive the benefits arising from the renouncement of locks as wells as conflict resolution. More detailed information about the performance of all three mechanisms will be provided in Subsection 3.1.

3 Prototypical Implementation and Benchmarking

As mentioned in Section 1, the presented prototypical implementation has been done in the context of a joint research project between the Hasso Plattner Institute and SAP. The used database system is based on proprietary SAP DBMS technology and is referred to as relational, column-oriented, in-memory DBMS in the remainder of this paper.

Our prototype ATP had to tackle the following architectural requirements: Multiple concurrent orders have to be possible and are to be handled without risking inconsistency. The prototype renounces the limitation of static buckets in current ATP systems, enabling distinctions in delivery date granularity on a per-demand basis.

The implementation we set out to build was to be optimized for columnar, in-memory storage [Hen10]. In order to provide fast order confirmation, our implementation also exploits parallel execution potential within the concurrency control strategies. It provides special views for analytical queries which allowed us to specify key figures upfront, yielding improved performance in analytical queries, particularly those important to create the time series for a candidate check. The application logic is implemented in Python, accessing the database with via its SQL interface. Computationally intensive parts have been ported to C++ and are imported as compiled modules.

3.1 Benchmarks

The benchmarks were conducted on a 24 core Xeon (four physical cores each comprising six logical cores) with 256 GB of main memory. This server was used for both, application and database, eliminating network performance as a factor in accessing the database. Furthermore, in our benchmarks we disabled logging of transactions to prevent hard disk transfer speed from posing as a bottleneck in our scenarios.

Our benchmarks were conducted on a dataset that was derived from a live ATP system of a Fortune 500 company. Today's ATP systems do not support analytics on top of ATP checks and only keep a short horizon of data as an active dataset and discard data after a few days. Thus, we generated a dataset spanning 3 years from a set of records covering 3 months of planning horizon. We ran our benchmarks on datasets ranging from 1 to 64 million active transaction items, where every order has 5 to 15 line items. Other dimensions were not scaled with the number of transactions, simulating different company sizes.

The immediate goal of our benchmarks is to show the feasibility of our prototype and proposed data structures. The exclusive strategy provides a comparison with existing systems from an algorithmic standpoint - it reflects how checks in modern ATP systems are conducted with the only difference being the data storage.

3.1.1 Dataset Size

The first benchmark serves as a proof of concept of the database architecture applied in our system. On a varying dataset size, the same operations are performed. In each run, 10 concurrent processes execute 400 checks. The critical part concerning the overall runtime of an ATP check is the aggregation query. The time spent in the application or to be exact the candidate check is supposed not to be affected by the dataset presuming a constant distribution of fact entries over the time. In this case, the size of the time series is determined only by the chosen granularity, which will not be changed so that the application part is constant.

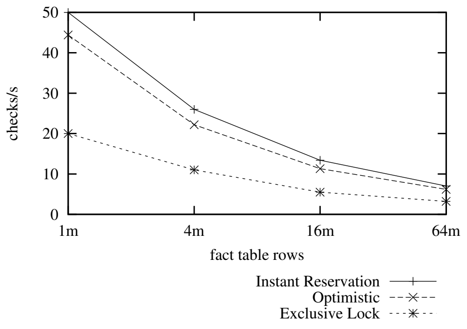


Figure 4: Varying dataset size

To conclude, the expected outcome of this benchmark is a linear relation between the dataset size and the query execution performance. Figure 4 displaying the throughput of checks explicitly reflects this trend. For the remaining experiments in this paper we work on 64 million records item which represents three years of operations in a large company.

3.1.2 Concurrency Control - Single Check

The following experiments directly compare the three concurrency control mechanisms introduced in Subsection 2.3. For this purpose, at first a single check is executed in isolation to evaluate the overhead added by the concurrency control. To recap, the exclusive lock only wraps the candidate check with the acquiring and releasing of a lock on product level. Both are atomic operations and do not involve database access. So, the overhead is supposed to be negligible.

In contrast, the optimistic approach requires a consistency check, which might induce a loop of conflict resolutions, with a maximum recursion depth though. The instant reservation mechanism neither uses locking nor consistency checks. Instead, it writes the promise first and has to validate it in the end. In case of a mismatch with the calculated delivery confirmation, an additional write operation to correct the promise is triggered. In Figure 5, the elapsed time split into read, write, and computation is illustrated. As expected, the optimistic check performs worst in this experiment, because it has to retrieve the recent fact entries to control the stock level after booking.

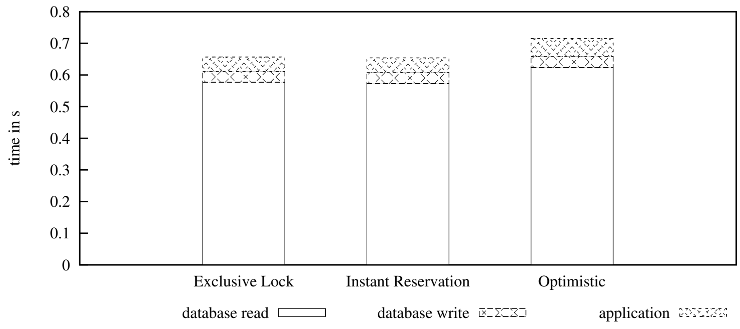


Figure 5: Single check

Since the inventory has not changed during the check and enough items were available, a conflict resolution has not happened. Neither a promise adjustment in the instant reservation run was necessary. Hence, the overhead is minimal for the two strategies that dispense with locking. The process of the exclusive lock is independent on the scenario and does not prolong the check anyway. Nevertheless, the scenario was not constructed to show the best case performance of the any strategy but rather to emulate the most common conditions with sufficient stock.

3.1.3 Concurrency Control - Throughput

After comparing single check times, the effectiveness of the presented techniques when facing parallel requests is measured, as it is the primary reason for putting so much emphasize on this topic. The prevailing KPI to assess concurrency control strategies is the throughput, in our case the number of accomplished checks per second. The setup for this experiment consists of 1000 checks to be executed on the machine specified above by a varying number of processes representing the degree of parallelization. In the first experiment, the checks refer to different products.

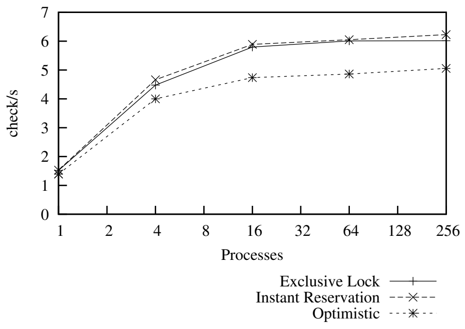


Figure 6: Disjoint CVC access

On the basis of the single check times and the hardware capacities, conclusions about the throughput can easily be drawn. The exclusive lock allows parallel requests on different products, so do the optimistic and the instant reservation approaches. Logically, the throughput scales linearly with an increasing number of processes. Since there is only a limited number of physical and logical cores, the increase of the curves flattens when the hardware is fully exploited, as can be seen in Figure 6.

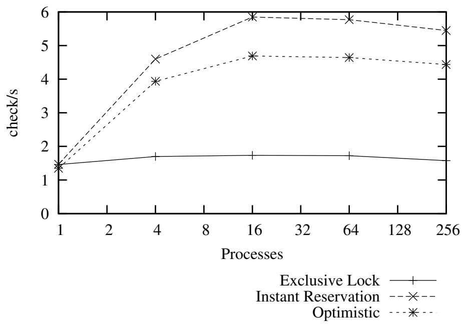


Figure 7: Single CVC access

For concurrent checks on one single product, another behavior is to assume. No matter how many processes are used to carry out the 1000 checks, only one process at a time can operate on the requested product in case of the exclusive lock. The logical consequence would be a constant throughput, which could be verified experimentally in our measurements, cf. Figure 7. This benchmark further gives evidence for the scalability of the two other alternatives in case of concurrent checks on one product. Those scenarios heavily benefit from locking-free concurrency control, whereas the exclusive lock enforces a sequential execution.

3.1.4 Write-Intensive Operations

Column-stores are particularly tailored for analytical queries. The workload of an ATP application is not limited to availability checks only but includes regular write operations as well. New orders and delivery promises have to be saved with every order. All new inputs and changes in inventory and production result in update and insert operations. The prevailing solution to handle a mixed workload consists of the division of the database into two parts, a read-optimized and a write-optimized store, the delta store, as briefly touched on in Section 1. The read-optimized store contains a snapshot of the database at a pre-established point in time. All incoming modifications are written to the delta store [KHK80]. The idea of organizing the data in two isolated stores dates back to the sixties. Early studies propose to regularly consolidate all modifications in the write-optimized store into the read-optimized store [SL76]. This so called merge process is the only procedure that modifies the read-optimized store. Since there is no need for maintainability or updatability, it can store the data most suitable for fast read access.

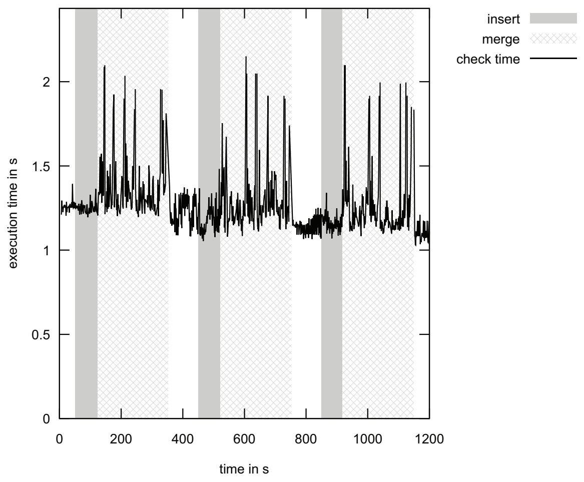


Figure 8: Performance during merge

In the delta-store, fast inserts can be enabled by not using the compression and keeping the data unsorted. The obvious effect is that ATP queries slow down, when the delta store reaches a certain size, since both stores need to be considered. In the light of this performance loss, a naive suggestion would be an eager merge policy minimizing the response times of analytical queries. However, the merge process itself is a complex procedure consuming a considerable amount of system resources. So, merge scheduling strategies have to deal with the tradeoff between merging often to keep the write-optimized store small and merging rarely to reduce the influence of the merge on the regular operation. To get an impression on how the merge process affects the ATP query execution, a long-term benchmark has been run on a dataset of 64M records. One process continuously executes ATP checks and another one inserts in cycles 1000 fact entries that are merged instantly to the read-optimized store. The curve in Figure 8 shows the total time spent on one check while the second process is in insert, merge, and idle phases and highlights the importance of a smart merge policy.

4 Business Implications

After describing the involved concepts and their prototypical implementation, this section lists possible business implications that could be realized by running the ATP application in a productive environment.

Parallel ATP Checks on Hot Products. Performing ATP checks on aggregates necessitates exclusive locking. As demonstrated in the previous section, we can omit exclusive locks in our prototype. This increases the throughput of simultaneous checks on the same product by using multi-core technology. The limit of performing simultaneous checks on single products is a problem, e.g. for high-tech companies when introducing new, highly requested products.

Changing the Temporal Granularity for Every Check. Since for every check the corresponding time stamp is saved, it is possible to change the temporal granularity for every single check. Thus, a check on hours or on weeks can be done in the same system without any modifications. This is not possible with aggregates as they can only operate on one initially defined temporal granularity.

Considering Product Attributes in the Check. The inclusion of additional attributes during the check is supported by the columnar data structure and has a significant business impact. Now, companies are able to include fine-granular product attributes in their checks for otherwise identical products. Examples are the date of expiry in the food industry or the quality of raw materials e.g. in the steel industry.

Analytics on Check History. The ability to do analytics on the history of ATP checks introduces two major advantages: on the one hand, a company can perform classical analytical tasks such as seeing which products were sold the most, which customer groups ordered which kinds of products in which time periods and so on. On the other hand, storing the complete history of ATP checks also including those checks which did not result in actual orders, makes an important source of information accessible: companies can see which products were highly requested, but were not ordered e.g. because not enough quantities were available. Furthermore, a company can see which are the most popular replacement products, e.g. which products were ordered in the end, although different products were initially included in the ATP check.

Instant Order Rescheduling. Promised orders and planned deliveries are based on a certain schedule of incoming products or planned production. However, often these schedules and plans turn out to be incorrect as products may not arrive on time or production goals cannot be met. As a consequence, already promised orders have to be rescheduled. Using aggregates, this is a time-intensive process as the relevant aggregates have to be calculated back to the point where the order in question was promised, so that the new delivery date can be calculated considering the changed stock projections. This operation can now be done significantly faster as all the relevant fine granular data is on hand. Including additional attributes such as prioritization of customers implies further functionality: A major customer's order that precedes other orders in its relevance can be protected from rescheduling activities incurred by unexpected changes to a product's stock level.

5 Conclusion

The real-time ATP approach presented in this paper does not only tackle performance bottlenecks, but also enables innovative features. On the technical side, we introduced the candidate checking algorithm and identified the instant reservation strategy as suitable concurrency control mechanism for executing an ATP check on an in-memory column-store. Based on a dataset with 64 million transactional records, we achieved a check time of 0.6 seconds. The dataset is based on the anonymized data from a Fortune 500 consumer products company and spans three years of operation. Our approach scales linearly with added CPU cores, even in hot-spot situations with checks against the same product. On the business side, we described possible business implications of our approach. To our knowledge, many ATP related systems do not track availability checks which did not result in orders and thereby lose extremely valuable information for planning purposes. Furthermore, rescheduling of orders is a time-intensive, static process without the possibility to include further requirements. Only these two aspects alone provide a significant benefit for companies.

The outlook of this paper leaves the safe harbor of well-established database concepts, prototypical implementations, and measurable results, but draws a vision of how companies could leverage analytical capabilities in the context of an ATP check in order to maximize their profits. On the one hand, let us assume that we can analyze the full history of a customer's ATP checks and resulting orders during every new incoming check. That implicates that we can calculate the probability of a customer still placing his order even if he cannot get the products delivered on his initially requested date. Therefore, we can derive a certain flexibility on the companies' side when to produce and ship an order without actually loosing any orders. On the other hand, we heavily discussed the consideration of additional attributes during the check throughout the paper. Another example for such an attribute could be the varying production costs for the different products over time. Even if companies sell products for the same price over a certain period of time, the real production costs vary heavily due to changing raw material costs, different availability and costs of labor, and changing component suppliers. Putting those pieces together, instead of just considering the available quantities during the check, a company could also include varying production costs and therefore present an availability date that aims at maximizing the profits.

References

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. ColumnStores vs. RowStores: How Different Are They Really? *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, page 967, 2008.

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. *Very Large Data Bases*, 1999.

[CGK01] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query Optimization In Compressed Database Systems. *International Conference on Management of Data*, 30(2), 2001.

[Dic05] Jörg Thomas Dickersbach. *Supply Chain Management with APO*. Springer, Berlin,Heidelberg, 2005.

[GS91] Goetz Graefe and Leonard D. Shapiro. Data Compression and Database Performance. pages 22–27, 1991.

[Hen10] Doug Henschen. SAP Announces In-Memory Analysis Technology, 2010.

[HLAM06] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance Tradeoffs in Read-Optimized Databases. *Very Large Data Bases*, 2006.

[KGT$^+$09] Jens Krüger, Martin Grund, Christian Tinnefeld, Jan Schaffner, Stephan Müller, and Alexander Zeier. Enterprise Data Management in Mixed Workload Environments. *2009 16th International Conference on Industrial Engineering and Engineering Management*, pages 726–730, October 2009.

[KHK80] V. J. Kollias, M. Hatzopoulos, and J. G. Kollias. Database maintenance efficiency using differential files. *Information Systems*, 5(4):319–321, 1980.

[KTG$^+$10] Jens Krüger, Christian Tinnefeld, Martin Grund, Alexander Zeier, and Hasso Plattner. A case for online mixed workload processing. In Shivnath Babu and G. N. Paulley, editors, *DBTest*. ACM, 2010.

[Mar04] Tim Martyn. Reconsidering Multi-Dimensional schemas. *ACM SIGMOD Record*, 33(1):83, March 2004.

[Pla09] Hasso Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. *Read*, 2009.

[Sc05] Michael Stonebraker and Ugur Çetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In *ICDE*, pages 2–11. IEEE Computer Society, 2005.

[SL76] Dennis G. Severance and Guy M. Lohman. Differential files:their application to the maintenance of large databases. *ACM Transactions on Database Systems (TODS)*, 1(3):256–267, 1976.

[SZ03] Rainer Scheckenbach and Alexander Zeier. *Collaborative SCM in Branchen*. Galileo Press GmbH, Bonn, 2003.