Improving the Scalability of Cloud-Based Resilient Database Servers*

Luís Soares and José Pereira

University of Minho
{los,jop}@di.uminho.pt

Abstract. Many rely now on public cloud infrastructure-as-a-service for database servers, mainly, by pushing the limits of existing pooling and replication software to operate large shared-nothing virtual server clusters. Yet, it is unclear whether this is still the best architectural choice, namely, when cloud infrastructure provides seamless virtual shared storage and bills clients on actual disk usage.
This paper addresses this challenge with Resilient Asynchronous Commit (RAsC), an improvement to a well-known shared-nothing design based on the assumption that a much larger number of servers is required for scale than for resilience. Then we compare this proposal to other database server architectures using an analytical model focused on peak throughput and conclude that it provides the best performance/cost trade-off while at the same time addressing a wide range of fault scenarios.

Keywords: Database servers; cloud computing; scalability; resilience.

1 Introduction

There is a growing number of organizations taking advantage of infrastructure-as-a-service offered by public cloud vendors. In fact, multi-tiered applications make it easy to scale out upper layers across multiple virtual servers as they are mostly "embarrassingly parallel" and stateless. The scalability, availability, and integrity bottleneck is still the database management system (DBMS) that holds all non-volatile state.

Although there has recently been a call to rethink databases from scratch, leading to NoSQL databases such as Amazon SimpleDB, this challenge is still being addressed by pushing existing SQL database server clustering to the limit. The simplest approach, since it doesn't require explicit support from the DBMS and addresses only availability, is failover using virtual volume provided by the cloud infrastructure such as Amazon Elastic Block Storage (EBS). A more sophisticated approach is provided by Oracle Real Application Cluster (RAC) [1], also backed by a shared volume, but leveraging multiple hosts for parallel processing. An alternative is a shared-nothing cluster with a middleware controller

* Partially funded by project ReD (PDTC/EIA-EIA/109044/2008) and FCT PhD scholarship (SFRH/BD/31114/2006).

built also with any off-the-shelf DBMS and allowing parallel processing with many common workloads. This is the approach of C-JDBC [2]. In addition, a certification-based protocol, such as Postgres-R [3], can further improve performance by allowing execution of update transactions by a single server.

The trade-off between scalability and resilience implicit in each of the of these architectures is however much less clear. Namely, different options on how update transactions are executed lead to potentially different peak throughput for a specific resource configuration, in terms of available CPUs and storage bandwidth, with a write intensive load. Moreover, state corruption, namely, upon a software bug can have different impacts in different architectures. Avoiding the severe impact on availability when the only resort is to recover from backups is very relevant. Answering these questions requires considering how each architecture handles updates and to what extent different components of the database management system are replicated independently, thus leading to logically and/or physical replicas of data.

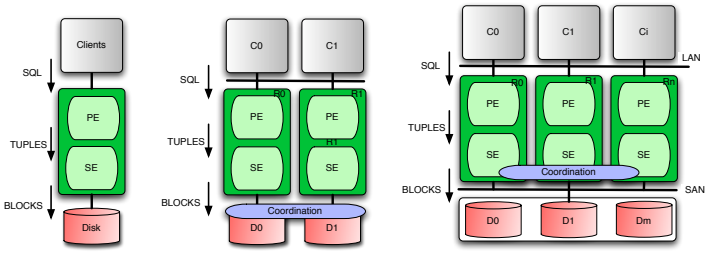
The contribution of this paper is therefore twofold. First, we propose Resilient Asynchronous Commit (RAsC), an improvement to certification-based database replication protocol that decouples effective storage bandwidth from the number of servers, allowing the cluster to scale in terms of peak throughput. We then re-evaluate different architectural aspects for clustering with an analytical model that relates each of the described architectures with resilience and scalability metrics.

The rest of this paper is structured as follows: In Section 2 we provide the background on clustering architectures. Section 3 proposes an optimization on one of the clustering architectures. Section 4 introduces the model. Section 5 compares different architectures within this model, justifying the relevance of the proposed contribution. Finally, Section 6 concludes the paper.

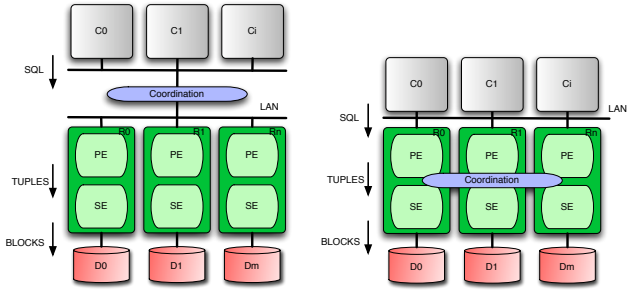
2 Background

Transaction processing in a relational database management system is usually regarded a layered process [4]. At the top, SQL is parsed. The resulting syntax tree is then fed to the optimizer, which uses a number of heuristics and statistical information to select the best strategy for each relational operator. The resulting plan is then executed by calling into the logical storage layer. In this paper, we use a simplified view of transaction processing as a two layer process as depicted in Figure 1(a): We consider parsing, optimization, and planning as the *Processing Engine* (PE) and logical and physical storage management as the *Storage Engine* (SE). This maps, for instance, with MySQL's two-layer architecture, with pluggable storage engines.

Note that assertive faults at PE and SE levels, that lead to erroneous results, have very different impacts. At the SE level, they may invalidate basic assumptions of physical layout and of transactional recovery mechanisms and lead to invalid data. This can only be recovered by taking the server off-line and, possibly, only by restoring from backup copies. Faults a the PE level will



(a) Standalone. (b) Shared Disk Failover (SDF). (c) Shared Disk Parallel (SDP).



(d) Shared Nothing Active (SNA). (e) Shared Nothing Certification-Based (SNCB).

Fig. 1. Standalone and clustered servers.

still be contained within transaction boundaries and can be recovered by undoing affected transactions, manually or from undo logs.

The key defining architectural decision of database server clustering architectures is the amount of sharing that occurs, which defines at which level coordination happens and what layers (PE and/or SE) are replicated or shared. This determines not only the resulting scalability and resilience trade-off, but the applicability of each architecture to an off-the-shelf database server. We now examine four representative architectures.

Shared Disk Failover (SDF). Cluster management software ensures that the DBMS server is running in only one of the nodes attached to a shared disk, often using a Storage Area Network (SAN). If the currently active node crashes, it is forcibly unplugged and the server is started on a different node. The standard log recovery procedure ensures the consistency of on-disk data, thus it is applicable to any DBMS. A variation of this approach can be built without a physically shared disk by using a volume replicator such as DRBD [5]. Otherwise, disk redundancy is ensured by a RAID configuration.

This architecture is thus targeted exclusively at tolerating server crashes and is often deployed in a simple two server configuration. As depicted in Figure 1(b), coordination exists only outside the DBMS ensuring that the shared volume is mounted exactly by a single server. It is impossible to use the standby nodes

even for distributing read-only load as cache coherence issues would arise if the volume was mounted by multiple nodes. Since replication is performed at the raw disk level, neither the PE or SE are replicated in updates and no tolerance to corruption is provided.

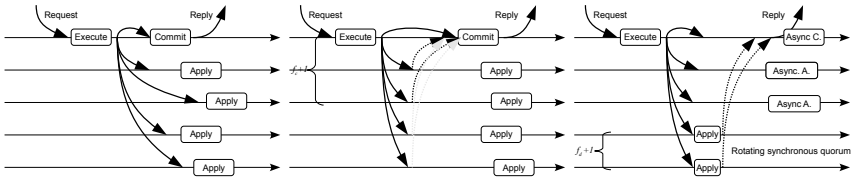
Shared Disk Parallel (SDP). Allowing multiple nodes to concurrently access the same shared storage requires that caches are kept consistent. In detail, the ownership of each block changes through time, in particular, whenever a write operation is issued. A distributed concurrency control mechanism is thus responsible to hand over the page to the issuing instance and no I/O is required in this process, even if the page is dirty. Reads are shared by having the owner to clone the page whenever a read request is issued. Flushing blocks back to disk is performed by only one replica at the time. As shown in Figure 1(c), coordination is thus performed within the storage engine layer. An example of this architecture is Oracle Real Application Cluster (RAC), which is based on the Oracle Parallel Server (OPS) and *Cache Fusion* technology [6].

This architecture is thus targeted mainly at scaling the server both in terms of available CPU and memory bandwidth, although it provides the same degree of fault tolerance as SDF, since most of the server stack is still not replicated in update transactions.

Shared Nothing Active (SNA). By completely isolating back-end servers, a middleware layer intercepts all client requests and forwards them to the independent replicas. Scalability is achieved as read-only requests are balanced across available nodes. Only update transactions need to be actively replicated on all replicas. The controller thus acts as a wrapper. It exposes the same client interface as the original server, for which it acts as a client. There is no direct communication between cluster nodes, as coordination is performed outside servers, as shown in Figure 1(d). A popular implementation is provided by Sequoia, formerly C-JDBC [2], which intercepts JDBC and is portable to multiple back-end servers.

The major scalability drawback is that update statements must be fully deterministic and have to be carefully scheduled to avoid conflict that translate into non-deterministic outcome of the execution and thus inconsistency. In practice, this usually means not allowing concurrent update transactions at all. This architecture is thus targeted at scaling the server in face of mostly read-only workload. By completely isolation back-end servers, it replicates all layers in update transactions and thus tolerates all outlined assertive fault scenarios in both PE and SE. In fact, a portable implementations such as Sequoia even supports DBMS diversity. In principle, it could even support voting to mask erroneous replies based on corrupt state [7].

Shared Nothing Certification-Based (SNCB). Active replication of update transactions in shared nothing clusters can be avoided by using a certification-based protocol. Each transaction is thus executed in the replica that is directly contacted by the client, without any *a priori* coordination. Thence, transactions get locally synchronized, according to the local concurrency control mechanism and only just before commit a coordination procedure is initiated. At this time,



(a) Regular. (b) Safe. (c) RAsC.

Fig. 2. Variations of the SNCB architecture.

the initiating replica multicasts updates using a totally ordered group communication primitive [8]. This causes all nodes to deliver the exact same sequence of updates, which are then certified by testing for possible conflicts. This leads to the exact same sequence of transaction outcomes that is then committed independently by each node. Although no commercial products based on this approach exist, there have been a number of related research proposals and prototypes [3, 9–11].

Since coordination happens between Processing and Storage Engines (Figure 1(e)), it is capable of performing fine grained synchronization of scaling with an update intensive workload. As a consequence of shared execution, this approach does not tolerate logical corruption, however, is tolerates physical corruption at the storage engine and disk layers. This is a very interesting trade-off, since such logical corruption can be corrected by undoing changes even while the system is on-line.

3 Resilient Asynchronous Commit

The SNCB architecture thus offers a very interesting trade-off: Since assertive faults at the PE can be corrected by undoing changes even while the system is on-line and it naturally copes with assertive faults at the SE level, it provides much of the advantages of the SNA architecture with a potentially better peak throughput.

Traditionally, certification-based protocols use asynchronous group communication primitives for handling message passing between the replicas in the cluster, as shown in Figure 2(a). Thus there is a chance that updates are lost in the situation that the originating server's disk is lost. To improve resilience one can resort to an uniform reliable or safe multicast primitive [8], that gathers a number of acknowledgments from at least $f_c + 1$ nodes prior to delivery (Figure 2(b)), where f_c is the upper bound on process faults. This ensures that a number of other nodes have stored the update in memory and unless the cluster fails catastrophically, it will eventually be committed to all disks [12].

Nonetheless, even if acknowledging transaction commit to the client awaits only for local commit to disk, existing proposals do not distinguish commits that are out of such critical path and will still force updates to disk. This poses an upper bound on database server scale-out, as storage bandwidth consumed by each

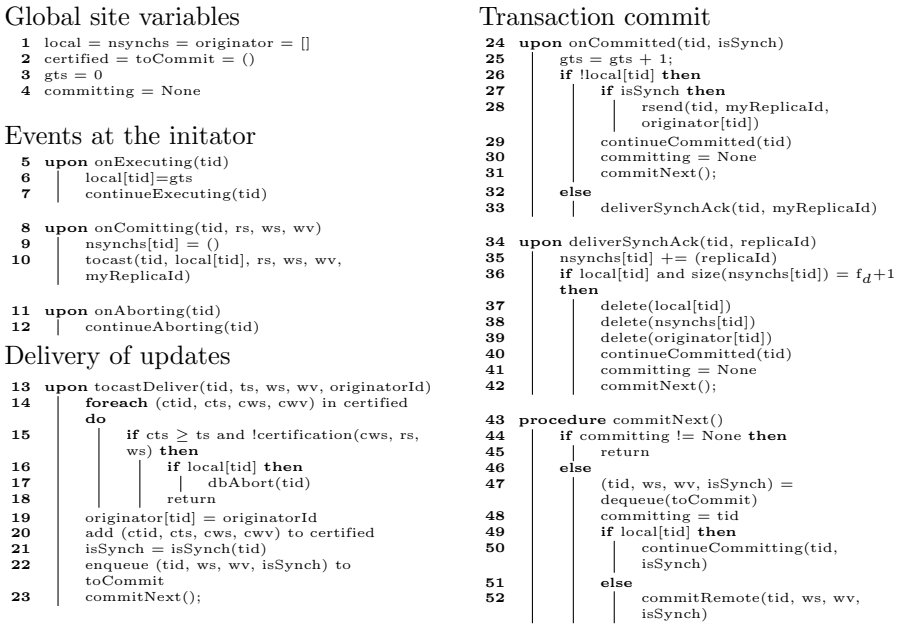


Fig. 3. Resilient Asynchronous Commit Protocol (RAsC).

replica grows linearly with the size of the workload being handled. Our proposal thus stems from the observation that the substantial storage bandwidth economy resulting from asynchronous commit [13] can also be obtained by sharing the burden of synchronous commit across a large number of replicas.

Moreover, the same mechanism should allow waiting for multiple disk commits such that scenarios with catastrophic failures can be handled. In detail, this means performing an asynchronous commit on $n - (f_d + 1)$ nodes (where f_d is the number of tolerated disk faults and n the number of replicas), and a synchronous commit elsewhere. Then we defer acknowledgment until synchronous commit concludes. The resulting protocol (RAsC) is shown in Figure 2(c), in which commit waits for a rotating subset of replicas to commit to disk.

Figure 3 details in pseudo-code the proposed Resilient Asynchronous Commit protocol in combination with SNCB. The initiator is the site in which a transaction has been submitted. Handlers, or hooks, are assumed to exist and are called by the DBMS Transaction Manager. A set of interfaces targeting this behavior has been proposed and several prototypes exist [14]. Nevertheless, these hooks are further explained in the next few lines. Before a transaction tid executes its first operation, the onExecuting handler is invoked. The version of the database seen by tid is required for the certification procedure. Since we are considering snapshot isolation, this is equal to the number of committed transactions when tid begins execution. If the transaction at any time aborts locally, onAborting()

is invoked and the transaction is simply forgotten. After a successful local execution, the onCommitting hook is called, causing the updates to be atomically multicast to the group of replicas. This ensures atomic and ordered delivery of transaction updates to all replicas, which happens on the tocastDeliver hook. After delivery, the certification procedure is performed and the fate of the transaction is deterministically and independently decided at every replica. Within this hook, the isSynch() function determines if a synchronous commit is meant to happen at the replica. The isSynch() function determines whether this replica is in the rotating quorum for this transaction.

The last operation in the hook is a call to the scheduler that issues execution/commit on the next certified transaction (commitNext). Whenever a transaction commit finishes, which happens every time the onCommitted hook is called, the version counter is incremented. For remote transactions it checks if a synchronous commit has been performed and if so, an acknowledge is sent back to the initiator replica, using a reliable send communication primitive (rsend). Execution resumes by letting the Transaction Manager know that it may proceed (continueCommitted hook), and by scheduling the next certified transaction to commit (commitNext). For local transactions, a call to the deliverSynchAck is performed. The deliverSynchAck hook is called every time the initiator receives a synchronous commit acknowledge from a replica, or once the initiator commit finishes (in this case the initiator acknowledges its own synchronous commit). Once all the required synchronous commits have been performed the continueCommitted hook is called and local execution may resume, which ultimately results in notifying the client that the commit succeeded. A final note about the myReplicaId, replicaId and originatorId. These identifiers are used to perform message passing, which may even be IP addresses, should the replicas reside on different machines, or any other identifier that uniquely addresses replica processes.

4 Analytical Model

To select the best architecture for different fault and workload scenarios, and to what extent the Resilient Asynchronous Commit protocol improves the SNCB architecture, we model the amount of computing and storage resources (i.e. CPUs and disks) required to handle a given load while tolerating a number of failures. Depending on the architecture chosen, there are additional parameters. For instance, in a shared-nothing architecture, we have n independent nodes. In general, the system cost directly depends on the following parameters:

1. aggregate computing bandwidth (C);
2. aggregate disk bandwidth (D).

An architecture is preferable if it allows us to tightly dimension the system such that there is neither excess C or D. Also, that it allows the system to be reconfigured in order to separately accommodate changing requirements.

Assumptions The following assumptions hold in our model. They are backed by assumptions already made in previous work (Gray et al [15]).

- Each transaction t is made of a number of *read* (n_r) and *write* (n_w) operations ($n_o = n_r + n_w$), and we consider read-only ($n_o = n_r$) and update ($n_o = n_w$) transactions;
- Read operations never block because they operate in their own snapshot version of the database [16], hence only updates conflict;
- Read and write operations are equal in disk and cpu bandwidth consumption ($d_w = d_r = d_o$ and $c_w = c_r = c_o$), take the same time to complete (t_o), and each transaction completes execution in a given time t_t ($t_t = t_o \cdot n_o$);
- The system load (tps) is composed of a mix of update transactions ($wtps$) and read only transactions ($rtps$). These are correlated by a w_f factor ($w_f = \frac{wtps}{tps}$). The number of concurrent transactions in the system (n_t) is derived from the workload ($n_t = n_{tw} + n_{tr} = (wtps \cdot n_w \cdot t_o) + (rtps \cdot n_r \cdot t_o)$)).
- The size of the database (s) is the number of objects stored and item accesses are uniformly distributed;
- Failures exist (f), but they never result in the failure of the entire system;
- n itself is the number of replicas in the system.

In contrast to previous proposals that model distributed database systems [17, 18], we focus on the availability of a shared storage resources (space and bandwidth) offered by cloud infrastructure instead of assuming that storage is proportional to number of allocated servers.

Resource Bandwidth We start by modeling the baseline (NONE) which is a centralized database monitor with no disk redundancy. Bounds on system parameters are established by the workload. In a centralized and contention-free system, the disk and CPU used, by a transaction t, are generically expressed using Equation 1 and Equation 2.

$$d_t = dr_t + dw_t = (n_r + n_w) \cdot d_o \tag{1}$$

$$c_t = cr_t + cw_t = (n_r + n_w) \cdot c_o \tag{2}$$

An improvement over the baseline system (NONE-R), in terms of disk faults resilience and read performance, is achieved using a RAID storage system (m disks providing redundancy and parallelism). The tradeoff lies in the extra disk bandwidth ($m - 1$) required to replicate blocks same data.

$$d_{t-none-r} = (n_r + m \cdot n_w) \cdot d_o \tag{3}$$

A different approach altogether, would be to use DRBD. This solution replicates data at block level and provides multi-master replication by delegating to a top layer software (a clustered file system like OFCSv2 or GFS) conflict detection and handling. Nevertheless, concurrent writes are handled but they are not meant to happen regularly at the DRBD level. Furthermore, when a database is deployed

on top of the DRBD, the replication is performed in a master-slave (hot standby) fashion. This imposes a limit to resilience, as the number of replicas cannot be higher than two ($n = 2$). Due to its current resilience limitations we find this architecture rather uninteresting, and will not be considering it from now on.

$$d_{t-sdf} = dr_t + n \cdot dw_t = (n_r + 2 \cdot n_w) \cdot d_o \tag{4}$$

$$c_{t-sdf} = cr_t + n \cdot cw_t = (n_r + 2 \cdot n_w) \cdot c_o \tag{5}$$

SDF limitations may be easily mitigated by architecting a system based on a distributed middleware approach, mostly like SNA. In this architecture, a middleware controller acts as the load balancer for reads and coordinator for writes. Database back-ends are registered at the controller. Reads are only performed at one replica, while writes happen everywhere. This approach is very similar to a RAID based disk mirroring strategy, but instead of handling raw blocks, logical data representations (e.g., SQL statements) are synchronized and executed at each registered database instance. Equations 6 and 7 model the resource consumption in this setup. Unfortunately, this approach has limited scalability when dealing with write intensive (or write peaks) workloads and non-deterministic operations.

$$d_{t-sna} = (n_r + n \cdot n_w) \cdot d_o \tag{6}$$

$$c_{t-sna} = (n_r + n \cdot n_w) \cdot c_o \tag{7}$$

SNCB mitigates the issues exhibited by SNA. We assume independent servers, acting as a replicated state machine on write requests and with perfectly balanced read requests. This is the case for certification based approach to replicated databases (e.g., the Database State Machine - DBSM). Given that in a DBSM setting each replica writes the same data on its local storage, the disk usage is described by Equation 8 (we assume that the database working set fits in main memory, so we disregard disk usage for read operations). On the other hand, the CPU consumption does not increase by a degree of n. In fact, the optimistic execution guarantees given a transaction t, it *executes* completely at any given replica and the others only *apply* t's changes. Consequently, remote updates only take a fraction of the original CPU execution regarding the write operations. This is depicted by the correlation factor k_{apply}, which captures the cost of *applying* the updates versus *executing* the original update operations.

$$d_{t-sncb} = n \cdot n_w \cdot d_o \tag{8}$$

$$c_{t-sncb} = (n_r + (1 + k_{apply} \cdot (n - 1)) \cdot n_w) \cdot c_o \tag{9}$$

An alternative cluster architecture uses a shared storage. We assume that such storage is a RAID unit of m disks. Since we are not accounting for messages delays nor network bandwidth consumption, the disk bandwidth and cpu bandwidth are the same as in the NONE-R and NONE, respectively.

Finally, for all of the above mentioned architectures, the aggregate C and D bandwidth consumption is calculated as a function of the incoming transaction

rate (tps). This is depicted by Equation 10 and 11, respectively.

$$C = c_t \cdot tps \tag{10}$$

$$D = d_t \cdot w_f \cdot tps \tag{11}$$

Resource Contention The aggregate CPU and Storage bandwidth consumption is driven by the workload (tps). Note that dependent on the workload is also the contention rate of the system. Therefore, the aggregate consumption must be calculated by taking into account contention. In [15], and under the same assumptions presented here, we may find that the generic expression for system contention rate (number of transactions waiting per second) is given by Equation 12.

$$tps_{wait} = (1 - (1 - (\frac{n_{tw} \cdot n_w}{2 \cdot s})^{n_w}) \cdot tps \cdot w_f \tag{12}$$

Except for the $SNCB$ and SNA architecture, this equation models perfectly transaction blocking. In $SNCB$, transactions tend to be in the system a bit longer than normal execution, due to the process of applying remote updates. As Equation 13 shows, n_{tw} increases, and the system becomes more susceptible to conflicts.

$$n_{tw^{sncb}} = w_f \cdot tps \cdot n_w \cdot t_o \cdot (1 + k_{apply}) \tag{13}$$

On the other hand, in SNA, transactions are set to execute sequentially by the controller which becomes a major bottleneck. Since only update transactions conflict and transaction execution is sequential, the number of transactions waiting in the system is given by Equation 14.

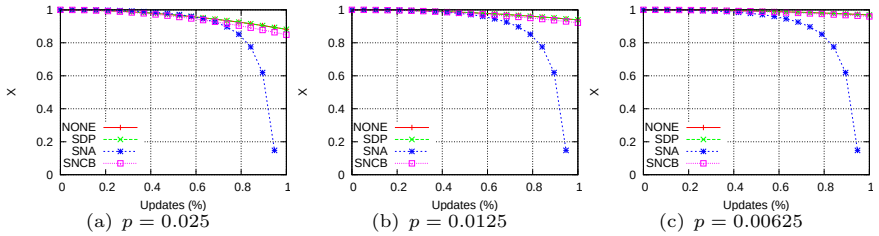
$$tps_{wait^{sna}} = \frac{(wf \cdot tps)^2}{(n_w \cdot t_o) \cdot ((n_w \cdot t_o) - (wf \cdot tps))} \tag{14}$$

Finally, contention has a negative impact on system performance, which means that the number of committed transactions per second is unarguably lower than the number of input transaction rate. As such, subtracting the waiting rate from the incoming rate, we get the overall system throughput (Equation 15).

$$tps_o = w_f \cdot tps - tps_{wait} + (1 - w_f) \cdot tps \tag{15}$$

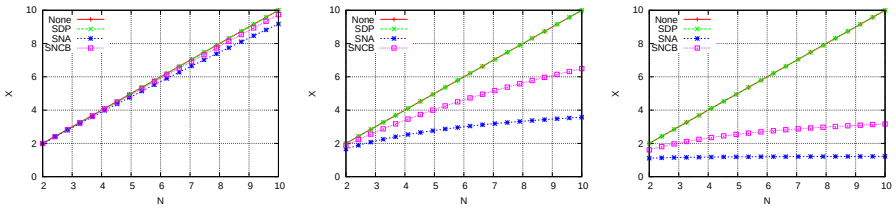
5 Evaluation

Strictly on the basis of resilience, one would probably choose the SNA architecture, after making the necessary changes to remove the single-point-of-failure introduced by the controller. In this section, we evaluate the cost of this option in terms of conflict scalability, i.e. how does it tackle peak write-intensive loads, and resource scalability, i.e. how does it take advantage of existing resources for performance.



(a) $p = 0.025$ (b) $p = 0.0125$ (c) $p = 0.00625$

Fig. 4. Impact of item conflict probability in throughput.



(a) $w_f = 0.01$ (b) $w_f = 0.2$ (c) $w_f = 0.8$

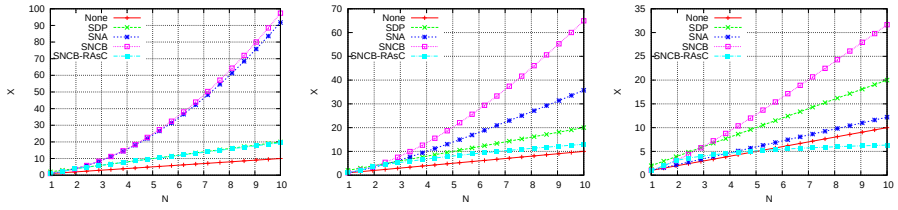
Fig. 5. Scalability of throughput with number of nodes, for different workload mixes.

Conflict Scalability We start by applying the contention model to determine how each architecture scales with different workloads with different amount of update transactions and updated items in each transaction. We do this by fixing an arbitrary offered load and then varying the ratio of update transactions from 0 to 1. Figure 4 show these with three different probabilities of single item conflicts (the p parameter). Previous experiments [19] indicate that TPC-C produces results comparable to Figure 4(c) and agree with the proposed model in terms of the resulting useful throughput in both shared-nothing scenarios.

The most interesting conclusion from Figure 4 is that the SNA approach exhibits a sudden saturation point with increasing number of update transactions, regardless of likelihood of actual conflicts. This precludes this architecture as a choice when there are concerns about possible write-intensive workload peaks leading to safety issues.

On the other hand, one observes that SNCB can approximate the performance of SDP. Neither exhibits the sudden tip-over point and thus should be able to withstand write intensive peak loads. Final notice, the NONE and SDP lines are a perfect match.

Node Scalability The next step is to apply the bandwidth model to determine how each architecture allows required resources to scale linearly with an increasing throughput. Therefore, we assume that computing bandwidth is provided in discrete units. To add an additional unit of CPU bandwidth one has therefore to add one more node to the cluster. This has an impact in shared nothing architectures, since each additional node requires an independent copy of data.



(a) $w_f = 0.01$ (b) $w_f = 0.2$ (c) $w_f = 0.8$

Fig. 6. Required storage bandwidth with number of nodes, different workload mixes.

Figure 5 shows the speedup that can be expected when adding additional nodes to the cluster. As expected, the SDP architecture should exhibit a perfect speedup, should the conflict probability be fixed, as happens for instance with TPC-C scaling rules. On the other hand, SNA shows that with anything other than an almost read-only load of Figure 5(a), this architecture scales very poorly. This is due to the fact that update transactions have to be actively executed by all nodes, and regardless of the contention effect described in the previous section.

Finally, Figure 5 shows that the SNCB architecture scales for low values of $w_f \times k_{apply}$. This means that efficiency when applying updates, for instance by using a dedicated low level interface, can offset the scalability obstacle presented by wrote intensive loads. Simple testing with the TPC-C workload and PostgreSQL 8.1, without using any dedicated interface, shows that $k = 0.3$, which is the value used henceforth.

Dedicated interfaces for applying updates have been implemented a number of times for DBMS replication. For instance, Oracle Streams and Postgres-R provide such interfaces for Oracle and PostgreSQL.

5.1 Disk Scalability

Figure 6 shows the aggregate storage bandwidth required to achieve the maximum theoretical scale up of Figure 5 and if possible, tolerating $f = 1$ faults. Namely, SDP tolerates only disk faults, regardless of nodes in the cluster. SNA with $n > f + 1$ tolerates f logical or physical corruption faults.

We now consider the following dilemma. Assume that one has 10 nodes in the cluster, and 20 disks. Each of the disks provides sufficient bandwidth for $1\times$ throughput. If one chooses the SDP architecture, it is possible to configure the storage subsystem with RAID 1+0 such that the exact bandwidth is achieved (i.e. 10 stripes, 2 copies). This allows $10\times$ the throughput. If one chooses SNCB, one has to opt for at most 2 stripes in each of the 10 copies. This is sufficient however for at most 5 nodes (from Figure 6(b)), which result in as little as $4\times$ the throughput (from Figure 5(b)). This is a 60% performance penalty.

Furthermore, one would be tempted to say that SNCB tolerates also f physical corruption faults with $n > f + 1$. However, certification-based protocols use asynchronous group communication primitives which jeopardizes that goal.

Nevertheless, by using Resilient Asynchronous Commit (RAsC) storage bandwidth is enough for the 10 nodes, thus for as much as $6\times$ the throughput. This is 50% more than the standard SNCB configuration. By executing synchronously only $f+1/n$ updates, this allows each of the nodes to use only $f+1/n$ of the previously required bandwidth, up to as much as $25\times$ with typical update intensive loads. This is shown in Figure 6(c).

6 Conclusion

In this paper we reconsider database server clustering architectures when used with a larger number of servers, in which cost-effectiveness depends on decoupling CPU and disk resources when scaling out. In contrast to previous approaches [3, 9–11], Resilient Asynchronous Commit protocol (RAsC) improves write scalability by making better use of resources with large number of servers on a shared storage cloud infrastructure without changes to the DBMS server, while at the same time allowing configurable resilience in terms of the number of durable copies that precede acknowledgment to clients.

Then we use a simple analytical model to seek scalability boundaries of different architectures and how shared resources in a cloud infrastructure can better be allocated. The first conclusion is that the currently very popular SNA architecture, although promising in terms of resilience should be considered very risky for scenarios exhibiting peak loads and write-intensive peaks.The second conclusion is that the SNCB approximates SDP in terms of linear scalability with moderate write-intensive loads and does not exhibit the risky sudden drop of performance with heavily write-intensive loads of SNA. The critical issue is the parameter k_{apply} in our model: The ratio of CPU bandwidth consumed when applying already executed updates. Finally, together with the proposed RAsC protocol, SNCB scales also in terms of storage bandwidth, especially with a relatively low number of assertive faults considered.

References

1. Ault, M., Tumma, M.: Oracle Real Application Clusters Configuration and Internals. Rampant Techpress (2003)
2. Cecchet, E., Marguerite, J., Zwaenepoel, W.: C-JDBC: Flexible database clustering middleware. In: USENIX Annual Technical Conference. (2004)
3. Kemme, B., Alonso, G.: Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In: VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (2000) 134–143
4. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall (2002)

5. Ellenberg, L.: DRBD 8.0.x and beyond: Shared-disk semantics on a shared-nothing cluster. In: LinuxConf Europe. (2007)
6. Lahiri, T., Srihari, V., Chan, W., MacNaughton, N., Chandrasekaran, S.: Cache fusion: Extending shared-disk clusters with shared caches. In Apers, P.M.G., Atzeni, P., Ceri, S., Paraboschi, S., Ramamohanarao, K., Snodgrass, R.T., eds.: Very Large Data Bases (VLDB) Conference, Morgan Kaufmann (2001) 683–686
7. Gashi, I., Popov, P., Strigini, L.: Fault diversity among off-the-shelf SQL database servers. Dependable Systems and Networks, 2004 International Conference on (28 June-1 July 2004) 389–398
8. Chockler, G.V., Keidar, I., Vitenberg, R.: Group Communication Specifications: a Comprehensive Study. ACMCS **33**(4) (December 2001) 427–469
9. Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. Distributed and Parallel Databases **14**(1) (2003) 71–98
10. Wu, S., Kemme, B.: Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In: ICDE '05: Proceedings of the 21st International Conference on Data Engineering, Washington, DC, USA, IEEE Computer Society (2005) 422–433
11. Elnikety, S., Dropsho, S., Pedone, F.: Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006. EuroSys '06, New York, NY, USA, ACM (2006) 117–130
12. Grov, J., Soares, L., Jr., A.C., Pereira, J., Oliveira, R., Pedone, F.: A pragmatic protocol for database replication in interconnected clusters. In: 12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), Riverside, USA. (2006)
13. Kathuria, V., Dhamankar, R., Kodavalla, H.: Transaction isolation and lazy commit. Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on (2007) 1204–1211
14. Correia, A., Pereira, J., Rodrigues, L., Carvalho, N., Vilaca, R., Oliveira, R., Guedes, S.: GORDA: An open architecture for database replication. Network Computing and Applications, 2007. NCA 2007. Sixth IEEE International Symposium on (12-14 July 2007) 287–290
15. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. Volume 25, 2 of ACM SIGMOD Record., New York, ACM Press (jun 1996) 173–182
16. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: ACM SIGMOD International Conference on Management of Data, New York, NY, USA, ACM (1995) 1–10
17. Bernabé-Gisbert, J.M., Zuikeviciute, V., Muñoz Escoí, F.D., Pedone, F.: A probabilistic analysis of snapshot isolation with partial replication. In: Proceedings of the 2008 Symposium on Reliable Distributed Systems, Washington, DC, USA, IEEE Computer Society (2008) 249–258
18. Elnikety, S., Dropsho, S., Cecchet, E., Zwaenepoel, W.: Predicting replicated database scalability from standalone database profiling. In: Proceedings of the 4th ACM European conference on Computer systems. EuroSys '09, New York, NY, USA, ACM (2009) 303–316
19. Jr., A.C., Sousa, A., Soares, L., Pereira, J., Moura, F., Oliveira, R.: Group-based replication of on-line transaction processing servers. In: Latin-American Symposium on Dependable Computing. (2005) 245–260