



Technology Overview: Solving SQL Scale-out

Version: April 17, 2011

Solving SQL Scale-out

In this paper, we examine the pressures driving the development of distributed databases and the SQL replication approaches being taken so far. We specifically examine the prominent issues surrounding MySQL clustering. However, the problem and commonplace replication approaches are ubiquitous across SQL platforms. Whilst the emergence of the 'NoSQL movement' promises a panacea for scale-out, it remains an alternate reality to SQL. As an exemplar, we discuss the benefits of GenieDB's integrative approach, extending the scope of an ultra-scalable NoSQL datastore to solve MySQL scale-out. We describe GenieDB's architecture that achieves immediate consistency within a fully replicated, self-healing database, and how this technology makes the same possible for today's SQL platforms.

Introduction

Over the past few years, "scaling" and "availability" have come to vie with "cutting costs" and "delivering new features" as major headaches for the CTOs of online businesses. With Moore's Law starting to lose steam, existing applications can no longer be scaled by simply buying faster hardware each year; instead, scaling an application has become an exercise in rebuilding it to utilise techniques such as sharding or non-SQL databases, while struggling to keep the existing live system going despite rising load, before users start leaving in frustration.

But the outlook isn't that bleak. "Clustering", a suite of technologies built around the concept of a group of independent servers co-operating to provide a service, has the potential to cut costs *and* provide scaling and availability in one fell swoop (sadly, however, they can't really help you deliver new features).

In this white paper we will illustrate the challenges of applying cluster computing solutions to online application problems, and how the latest technologies address those challenges.

But don't we already do clustering all the time?

Simple clustering has been a part of online application deployment since the very early days of the Web; a cluster of Web servers behind a load balancer is undoubtedly the "reference architecture" for an online application. But this is purely because Web servers are easy to cluster, as they do not store any information that changes in real time; instead, they refer to a database.

As the load on the system grows, more and more web servers can be added, and more and more Internet connections provided (through a set of redundant routers and load balancers), but the database soon becomes the bottleneck.

Cluster the SQL database

The first steps for most applications facing this issue are usually to deploy an "elastic cache" such as *Memcached*¹ and/or MySQL replication². Both of these approaches take advantage of

¹<http://en.wikipedia.org/wiki/Memcached>

²<http://dev.mysql.com/doc/refman/5.0/en/replication.html>

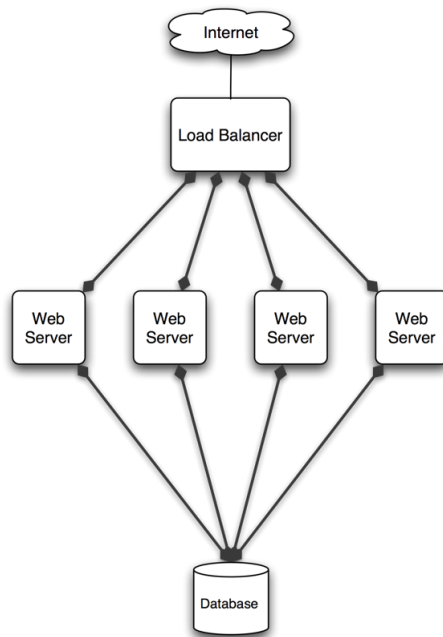


Figure 1: Standard Web Hosting Cluster.

the same premise - that the vast majority of database queries are reads.

An elastic cache creates a caching cluster, that your application uses to cache the results of SQL queries; the more servers you add to the cache cluster, the larger the cache is, and so more queries can be handled from the cache rather than needing to actually put load on the database. However, every UPDATE, INSERT, or DELETE has to be applied to the actual database as well as updating the affected records in the cache cluster, so writing to the database eventually becomes a bottleneck. Also, restarting the application after a power outage or other catastrophic event can become difficult; when full user load hits the web servers with an empty cache, every SELECT will have to be passed to the database - and if it was capable of taking that level of load, then you wouldn't have needed the cache in the first place! The application must be changed to correctly consult both the cache and the database layer in every place it does an SQL SELECT query, and to update or invalidate entries in the cache in every place it does an SQL UPDATE, INSERT, or DELETE command; this is relatively easy for queries that just select or update a single row by primary key, as that maps easily to the hashtable-like model of current elastic caches, but aggregate queries and bulk updates may require substantial work to fit into the cache model.

MySQL replication also improves read performance, but in a different way. A replicated MySQL cluster has a single database server known as the "master", and any number of "slave" servers. All UPDATE, INSERT, or DELETE commands are performed on the master, but the master keeps a "log" of these database-modifying SQL commands performed upon it, which the slaves pull from the master and apply locally, thus making them "follow" changes on the master. This means that SELECTs - the bulk of the load on the database - can be spread evenly across the slaves; and if each slave can service N queries per second, then M slaves can service N*M queries per second, so you can just add more slaves to increase your query capacity. However, again, the application must be modified; it must now maintain two separate database connections, one to the master for updates and one to a slave (chosen by some load balancing technique) for queries. If an update is ever accidentally sent to a slave, then that slave becomes

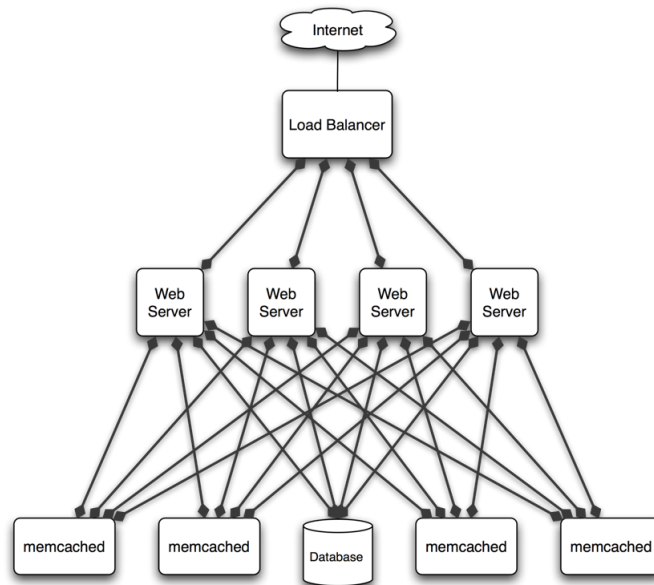


Figure 2: Scaling the database with an elastic cache such as *Memcached*

inconsistent with the master, and replication will silently fail - the slave will continue to run, but it will no longer receive updates from the master, so will become more and more out of date. Indeed, even in correct operation, slaves will become out of date when under high load, as the process of obtaining and applying updates from the master has to compete with other demands on the resources of the slave and master servers; while, in simple benchmarks, MySQL replication performs well under load, in practice it does so at the cost of providing increasingly incorrect data to clients. Applications have to be examined carefully to ensure that if the application ever assumes the result of an update will be reflected immediately in subsequent selects, those selects will have to be applied on the master rather than a slave, further concentrating the load on the one component of the system that cannot be easily expanded.

Both of these solutions involve changes to the application, and neither of them help much with availability, as failure of the master database server will make the entire system unable to process writes. MySQL replication offers the possibility of promoting a slave to become the new master should the master fail, but this is almost impossible to do in practice unless there is only one slave, as every other slave will need to be blanked and reloaded with a verbatim copy of the new master's tables in order to be in synch with it when replication is started; this will usually result in hours of downtime.

The next step systems take, if their update load becomes too high for any one master database to shoulder, is *sharding*, where some or all of the tables are split across multiple database servers, with some records on each. This means that the load - both queries *and* updates - is spread out across the cluster. The sharding is often based on a key such as a user ID, so that all rows of all tables relating to a particular user are on the same database server; this is because SQL databases rarely offer a practical way of doing joins between tables that are on different servers, so all rows that are likely to need to be joined together must end up on the same server. This requires extensive modifications to the application, as the application must now decide which databases to connect to depending on query or session parameters such as a user ID, while data that is not stored per-user (global site content, shared information such as forums,

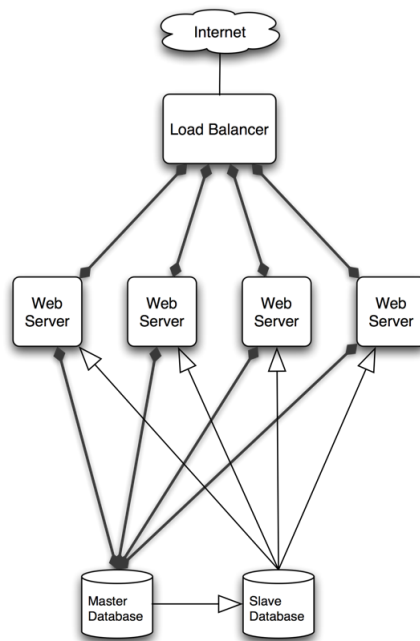


Figure 3: Scaling the database with master/slave replication.

advertising inventories, activity logs, public details of *other* users, and so on) often then need their own database connections to access the correct servers. Very large systems may combine all three approaches.

Replacing the SQL database

All of the above techniques are made difficult because SQL is fundamentally hostile to clustering. A single SQL command could potentially update every row in a table; a single SQL query could potentially read every row of *several* tables. This means that *distribution* - the fundamental technique behind the elastic cache and sharding approaches - can only easily help with a limited (but thankfully common!) type of update or query; those that only affect small well-defined sets of rows. This is because distribution involves scaling by splitting your database into small chunks and storing each chunk in a different server; if a query or command will involve more than one row, then the application has to decide which chunks might be involved and make sure the query or command is sent to every applicable server, and that the results are correctly merged together into one result. In effect, the application has to start doing some of the work normally handled automatically by the SQL server.

Similarly, *replication*, the fundamental technique behind MySQL replication, suffers from the limitation of SQL that updates are not deterministic; the effect of `UPDATE accounts SET balance = balance + 55 WHERE id = 23` depends on the *existing* balance of the account; and that's just the simplest case (what about `UPDATE users SET free_space = quota - (SELECT SUM(size) FROM images WHERE images.owner = users.id)` and let alone, `UPDATE images SET last_modified = NOW()`?). This means that the updates have to be applied in exactly the same order to every slave, starting from exactly the same initial state; even one small difference might cause subsequent updates to have increasingly different consequences. This makes SQL-based

replication systems fragile, increasing administration costs and downtime.

Because of this, there is growing interest in building “NoSQL” data stores. There are a number of open-source projects starting to explore these ideas such as CouchDB³ and Mnesia⁴, and hosted “cloud” services using non-SQL interfaces have appeared, such as SimpleDB⁵ and the BigTable⁶ storage in Google App Engine⁷.

These technologies offer much better scalability and availability than SQL-based systems, but at a cost; your application has to be substantially modified to use their special APIs (which developers must be retrained to use), but on top of that, the entire data architecture usually has to be reconsidered. The design of a database schema is always a tradeoff between the actual requirements of the actual problem - “products have an ID, name, a price, a description, a photo and a count of the number of times it is viewed, so we shall create a products table with those columns” - and the performance characteristics of your database - “updating the view counter in a product every time the product page is generated causes unacceptable delays due to contention over the row lock, and the presence of a BLOB field for the photo makes every operation on the table slower, so we’ll store the photos and the view counts in separate tables”. Since the new cluster database systems work in radically different ways, they tend to have radically different performance characteristics (for example, JOINS may become much more expensive than in conventional databases), and as such, require different approaches to how data is structured.

But perhaps the most significant change required in your application is usually a loosening of consistency, as found even with MySQL replication. Replication is an attractive approach, since it provides simple scaling for queries *and* increased availability by virtue of everything being stored in more than one place so the system can potentially continue to run after losing a server, but it suffers from a fundamental tradeoff: an update command must either avoid reporting completion to the application until every server has confirmed that it has processed the update (known as “synchronous replication”), which may take an arbitrarily long time if any server is overloaded or, worse, temporarily unavailable; or the update command can report success as soon as the replication of that update has been ‘started’ (eg, it has been written to the log on the master), and the application must come to expect that a select immediately after an update may not return the new state of the record. In very busy systems the replication “lag” may run to minutes or even hours, in which case a vicious circle can set in: the user goes and changes their details, but then when they see their own profile, they still see the old details. Wondering what they did wrong, they go back to the settings page and issue the change again, thus adding further load.

This “eventual consistency” model is popular because synchronous replication is slow and intolerant of temporary server or network failures, but it places a significant burden on the application developers to build their system to cope with it; doing so in all but the simplest of situations is a specialist skill, and one that few people have had time to learn (so those few can charge a hefty price for their services).

GenieDB

Our team started as a group of those specialists charging hefty prices for their services, but we quickly became frustrated by the limited tools available to us. MySQL replication was fragile and awkward, due to being a clustering system built on top of a system that was never designed

³<http://en.wikipedia.org/wiki/CouchDB>

⁴<http://en.wikipedia.org/wiki/Mnesia>

⁵<http://en.wikipedia.org/wiki/Amazon.SimpleDB>

⁶<http://en.wikipedia.org/wiki/BigTable>

⁷<http://en.wikipedia.org/wiki/Google.App.Engine>

to do it. Elastic caching is great, but requires application changes and cannot always be applied due to bulk queries. Non-SQL data stores offered the promise of a brave new world, casting away the limitations of past technology, but throw the baby out with the proverbial bath water - and choosing one involves major tradeoffs, as they all have significant downsides; they do not yet feel like well-balanced general solutions, more databases built for very specific types of problem (while practical applications usually involve a mix of different problems).

And so, the GenieDB Datastore was born. Starting from scratch, we merged the best of different data clustering technologies to create a hybrid solution, then parameterised the algorithms to allow selection of different tradeoffs to suit different problems on a per-query or per-update basis. We started by providing our own client API, then once the Datastore was mature, developed a MySQL pluggable storage engine that adapts our API to the MySQL table handler API, allowing Datastore tables to be accessed via SQL, so that our innovative back-end storage technology can be used by existing applications with little or no changes.

Replication

The Datastore is a replicated database; every server in the cluster holds a full copy of the entire database, and there is no central master, as any server can replicate a change to all the other servers. However, we have addressed the three major problems with this style of replication.

Immediate consistency

Making every update wait until it has replicated to every node would provide unacceptable response times for users, but the alternative - eventual consistency - makes life difficult for application developers, and can produce disconcerting behaviour for users.

However, this problem is notably absent in distributed systems such as sharding and elastic caching, so we found a way to provide the best of both worlds - by combining the two.

When a record is written to the Datastore, it is written into a sharded “consistency buffer”, similar to an elastic cache, *and* replication of that record is triggered in the background, then a successful response is returned.

If a query for that record is issued, then the consistency buffer is first checked - which will have the new version of the record, even while replication is still in progress.

Since the consistency buffer is treated as a cache, then failures of servers do not threaten the eventual consistency of the on-disk databases. A server failure will temporarily cause records that would have been covered by the server to have eventual rather than immediate consistency, but we still have eventual consistency in that case.

Since our underlying replicated datastore is designed for very low latency, the consistency buffer does not actually improve our performance as a cache; indeed, having to communicate over the network to the correct consistency shard slightly harms our performance, in exchange for providing immediate consistency. Therefore, for queries that can tolerate eventual consistency, we provide the option of bypassing the consistency buffer if required.

And since the consistency buffering is *built into* our client library, the application doesn't need to worry about it as it would with a conventional elastic cache - so the risk of bugs introduced by the application failing to correctly update the cache when an update occurs is removed.

Bandwidth utilisation

In a MySQL replication system with ten slaves, every update goes once to the master, then is fetched ten times - once by each slave. As the number of slaves rises to deal with greater and greater read load, the amount of network bandwidth used for each update rises too.

Our Datastore avoids this issue by using broadcast where possible. A group of servers connected to the same LAN will use the same amount of bandwidth per update as the update is broadcast to every node, using a reliable broadcast protocol. Where multiple LANs are involved (eg, for systems spread over several datacentres), then multicast can be used if the routing network supports it, or broadcast can be used within each LAN with a single unicast message per update sent between LANs, with the recipient of these unicasts on each LAN then broadcasting them on to the other nodes. Although some minor administrative messages sent once per second or less will use bandwidth that depends upon the number of servers in the cluster, the update messages will not, allowing scaling to large numbers of servers.

Flow control

One issue with asynchronously replicated databases is that, under a high write load, writes can pile up in queues faster than the slaves can process them. This means that the “replication lag” - the time difference between an update being issued and it being visible on all the servers, in an eventually consistent system - is potentially unbounded. It is possible to overload the system with writes so that the replication lag increases constantly as time passes and the queue worsens, until eventually queue storage is full and the system grinds to a screaming halt.

Our multi-master reliable broadcast-based replication ensures that writes are quickly sent to all the servers, and queued there, rather than being logged on a master server until slaves come and fetch them. The servers keep this queue in memory to reserve disk bandwidth for actual writes to the database, but this means the space available for the write queue is limited by available RAM.

However, our Datastore monitors the queue usage and total replication lag on each server and actively manages the write load to keep both queue usage and replication lag bounded. If either appears to be growing, write operations will start to sleep for a few milliseconds before proceeding, and that sleep will grow until the queue length and replication lag stabilise. The intelligent feedback algorithm automatically tunes itself for the parameters of your hardware and your application, so does not require manual adjustment; the result is that servers never run out of queue memory, and the replication lag that becomes visible when consistency buffer servers become unreachable is never worse than a few seconds, even under heavy load.

Record coalescing

We sought to generally reduce the requirements for applications to base their schemas around the performance characteristics of their database technology, where possible. In particular, we made it possible to have two views of the database schema: a logical view (as seen by the application) and a physical one (as seen by the actual storage system, and so actually influencing performance).

Our initial implementation only supports horizontal coalescing. Under this scheme, multiple logical tables can be assigned to a “table group”, indicating that they share the same primary key and should actually be stored as one “supertable”. This has no directly visible effect on those tables - but it acts as a hint to the Datastore that records from those tables with the same primary key will often be read or written together. When a record is requested from any

of those tables, the entire "superrecord" will be read in one unit (as it is stored contiguously on disk wherever possible), and all of it stored in the cache; any subsequent requests for any of the records from the same superrecord will then be serviced immediately from the cache.

Similarly, multiple writes to records in the same superrecord will be automatically merged into a single compound update, as the work queue on each server is not just a FIFO queue. The queue is indexed on superrecords; if an update to a record comes in and there is an existing update to a record in the same superrecord, then the new update is just merged in.

This allows the application to be written in terms of small logical records that can be updated independently, while disk access is done in terms of large physical tables, combining the best of both worlds.

Flexible semantics

Every operation in our native API accepts a set of flags, selecting different options for the desired semantics, because we feel that with the wide array of available tradeoffs in distributed data storage techniques, enforcing one needlessly limits a database to a certain type of application (or, worse, just a certain *part* of an application).

For example, our *GDSset* operation (which is responsible for updating records) has no less than four different modes of operation, as a result of enabling independent usage of two parts of the algorithm, with a third one in development to bring the total up to eight different modes - all, of course, with sensible defaults chosen to be appropriate in most situations.

Firstly, the use of the immediate consistency buffer can be disabled. Although most queries benefit greatly from immediate consistency, some might be perfectly well handled with eventual consistency - relatively static site content, or bulk loads of millions of records, for example. Disabling immediate consistency for bulk updates prevents needless flooding of the cache, pushing useful data out (and thus potentially harming immediate consistency of records that benefit from it), and also allows the bulk update to run much faster due to the removal of the need to connect out to consistency shard servers for every updated record.

Secondly, some updates need rigorous protection from primary key or unique index clashes. Independent of the distributed algorithm selection, *GDSset* has flags to select whether calling it on an existing row or one that clashes with another on a unique index is an error or causes the row to be updated, but by default, such a check is only applied on the local database (with primary keys also being checked in the consistency buffer) - meaning that updates that cause a unique index clash can both succeed if they occur in quick succession, or two attempts to create a unique new record with a given primary key may both succeed, yet the former is eventually overwritten by the latter as they meet in the cluster (the same one will "win" as the final state of the record on every server, due to the global timestamping system, which ensures every record has a unique timestamp, and the highest timestamp always wins). This can cause a problem in parts of an application such as signup, where it is an error to use an email address that is already in use; to rule out the possibility of clashes, we provide the optional "global check" flag to *GDSset* that causes it to become a two-stage process; the update starts by broadcasting to every server to reserve the new primary key value, and every server then responds to the client, either confirming the reservation or reporting that it has clashed with an existing reservation or actual record. If all the reachable servers confirm the reservation, then the actual update is broadcast, otherwise failure is reported back. This means that such rare special updates *can* be protected by the reservation algorithm, and bear the resulting latency cost, while updates that require low latency but do not run a risk of key collisions being a problem can choose a different tradeoff.

In general, we have tried to offer the widest range of semantics applicable to every operation - but with sensible defaults that will offer correct behaviour in all situations, leaving the use of finer-tuned semantics as an optional facility available to more advanced developers who have been trained in the tradeoffs involved; and which can be tuned by simply changing flags passed into our client library's operations, rather than requiring restructuring of the application.

Healing

Aware of the fragility of MySQL's replication, we designed the Datastore to be much more flexible. Every update is assigned a timestamp, unique across the cluster and steadily increasing. Every record stored on disk also stores the timestamp at which it was last updated. If an update appears that has a lower timestamp than that on the record on disk, it is ignored, otherwise it is applied; this mechanism ensures that, no matter what order two updates arrive in, the same one will consistently "win" in the long term. Although this is important in that it allows the Datastore to operate without a single central master to order every update consistently, it also allows the system to heal reliably. When a server fails and then returns, it looks at its on-disk copy of the timestamp of the update it last completely processed, and asks another server for the changes it has missed since then, while also listening for new broadcast updates. This means it can quickly and automatically catch up with the rest of the system, no matter what state it starts off in. Even if a network failure splits the cluster into two or more partitions, when connectivity is restored, the diverging states of the partitions can reunite by having each partition re-broadcast updates it experienced during the partition to the servers in the other partitions.

Also, we made the triggering of such replays of missed updates occur automatically whenever outages are detected, and made servers run an arbitrary administrator-supplied shell script whenever they detect that their local data is no longer in synch, or is back in synch after an automatic catch-up, so that servers can remove themselves from load balancer pools or otherwise avoid servicing requests based on outdated data.

Data model

Increased ease of implementing clustered storage isn't the only reason that many are breaking away from SQL; the structure of SQL tables, with fixed pre-declared columns, can be awkward in some applications - particularly object/relational mapped scenarios. Therefore, rather than enforcing a fixed column schema per table, the Datastore represents each row as an arbitrary list of named fields; each row in a table could, in principle, have a different set of fields, but in most applications, we predict that records in any given table will generally have a common set of core fields, then a number of fields that are optional and only appear in some records. If a record lacks a field that is indexed in the table in question, then that record simply does not appear in the index.

The MySQL storage engine

We designed our NoSQL API and its levels of tunable semantics so that it would be easily adapted to the MySQL table handler interface. This enabled us to develop a relatively simple wrapper to our client library that could be plugged into MySQL, making Datastore tables visible as MySQL tables. But being Datastore tables, the same table can be made visible on multiple MySQL servers - allowing you to scale MySQL by having a large number of servers, each seeing their own local copy of the tables, kept automatically in synch.

The issues with applying replication and distribution to an SQL database are avoided by not applying them at the SQL level, but at the record level; MySQL handles the actual queries and updates, and just tells our storage engine which rows it wants to read, insert, update, or delete.

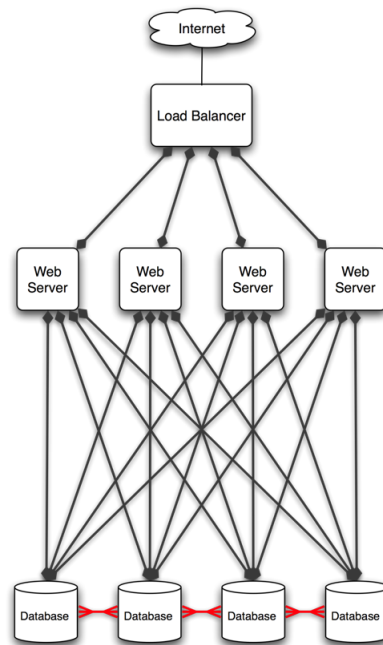


Figure 4: Scaling the database with GenieDB replication

This brings the best of both worlds; continuing with our philosophy of providing sensible easy-to-use defaults while allowing experts to use more advanced interfaces where required, our anticipated usage plan is that the MySQL plugin will quickly and easily allow existing applications, or applications written by developers trained only in MySQL, to reap the benefits of the Datastore; but if specific parts of the application require even more performance, then they can be modified as needed by sufficiently trained developers to use the native Datastore API.

GENIEDB®

31920 Del Obispo
Ste 260, San Juan Capistrano
CA, 92675
+1 (855) GENIEDB

Units 3-4 Orchard Mews
42 Orchard Rd
London N6 5TR
+44 208 347 5700

info@geniedb.com

GenieDB Inc.