Tips on database tuning...

SIZE MATTERS

Rasmus Resen Amossen • http://rasmus.resen.org • July 2011

Ironically, one of the worst things a database can be exposed to is data

WHY BOTH SIZE AND ORDER MATTER

The choice of data types for table columns may have significant impact on performance. More than you might think. This document describes why, and gives some concrete advice for unloading your database.

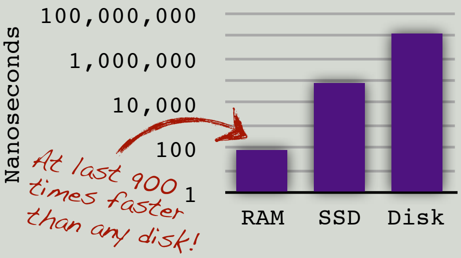
If you are impatient, skip all of this text and go straight to either the conclusions box to the right or the more elaborate Lessons Learned section on the last page, follow the advice there, and live happily ever after. Sort of. However, I recommend reading on, to get a better understanding of what is going on.

Let's first take a quick glance behind the scenes of the storage layer in common database engines, then explain the mechanisms behind the advice to come, and last, show some concrete experiments supporting the theory.

The storage layer - a quick overview

In most commonly known databases, tables are stored row-wise, and rows from the same table are physically organized in *pages*, typically of the size of an 8 kb disk block. The pages are, not surprisingly, stored on disk for persistency, that is, to prevent data loss when someone from the cleaning staff pulls out the server's power plug to make room for the vacuum cleaner's ditto. An access to a disk block is typically referred to as an I/O (short for Input/Output).

Unfortunately, disk access is extremely slow compared to RAM access. Fetching a random disk block from a 7200 RPM SATA disk takes roughly 13 milliseconds (or more than 40 million CPU clock cycles), whereas random access to RAM takes about 83 nanoseconds (or about 250 CPU clock cycles). So, the disk is around 160,000 times slower than RAM. Think about it. 160,000 times slower! A fast SSD has a seek time of 0.08 ms, but that is still more than 900 times slower than RAM.



CONCLUSIONS

✓ Random primary keys such as random GUIDs perform horribly. Consider using a sequential integer instead.

✓ Reduce row widths and table sizes as much as possible. Narrow rows perform better than wide rows - even with compression.

✓ Cache high-level data structures in a fast memory cache to ease the load on the database.

For the same reason, databases use a RAM buffer to cache data pages and reduce the need for time expensive disk access. Whenever a query needs to read a table row, the database first looks for its corresponding page in the buffer. If the page is not found there, we have a “buffer miss”, and it will then have to be fetched from disk, thereby increasing the time to find it with a factor between 900 and 160,000. After being fetched from disk, the page is stored in buffer (possibly replacing another buffered page) to speed up subsequent access to it. The term “buffer hit ratio” describes the fraction of pages needed by a query that can be fetched directly from buffer when needed. I believe we are ready to conclude: if a query has a low buffer hit ratio it will generally perform horribly.

So, can we do anything to increase the buffer hit ratio of a query, besides increasing the buffer size?

Actually, we can do a lot, but this text will mainly focus on two things: decreasing the row widths and choosing a good row ordering, which also turns out to have a significant impact on the buffer hit ratio.

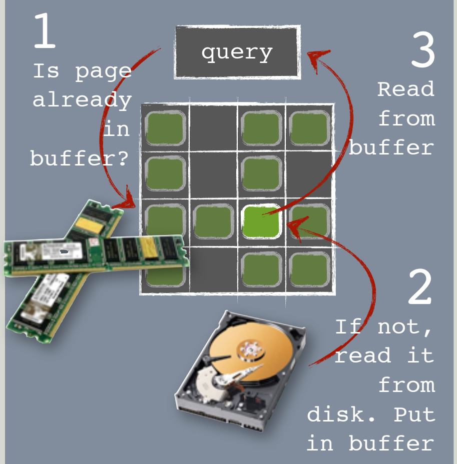
Less is more

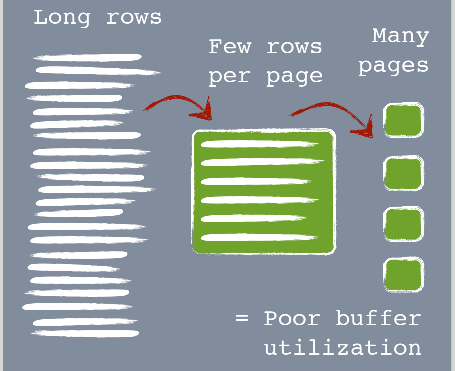
Intuitively, more narrow rows can fit into a single page than wide rows, and therefore a table with narrow rows can be stored in a smaller amount of pages than a table with wide rows. Also, the probability of any given page to exist in the buffer when needed becomes larger for small tables, so the narrow-row table (with narrow columns in terms of small data types) will likely give a higher buffer hit ratio and perform better than a wide-row table. The practical experiments shown later support this theory.

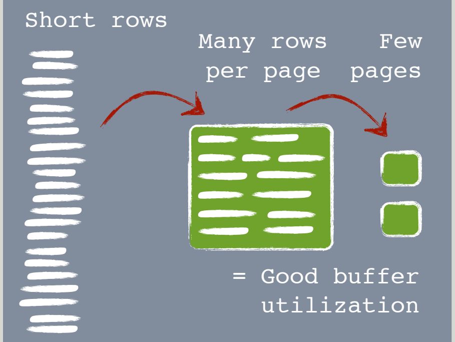
Some databases offer various kinds of compression, e.g. row compression and page compression. By compressing the data we almost obtain the same effect as choosing small data types. But the total size of a compressed table (independent of compression type) will likely still be smaller for narrow tables than for wide tables, so the rule of thumb still holds: always choose small data types!

Before diving into some practical experiments that illustrate this size-matters theory in practice, let's have a look at how rows are actually located when needed.

THE BUFFER


1
Is page already in buffer?
query
3
Read from buffer
2
If not, read it from disk. Put in buffer





Shorter rows, less space

The database uses a page buffer to decrease the need for costly I/O. Intuitively, the narrower rows, the fewer pages are needed to store a complete table. And the larger the probability is that any given row is already in a buffered page when needed.

SAME THING, DIFFERENT TERMS

One *buffer* holding multiple *buffer pages* or *buffer slots*, or one *buffer pool* holding multiple *buffers*. You may see different terms depending on where you read.

Wasting space may decrease speed

Indexing

A specific row can be located in multiple ways: either by simply traversing each and every page related to the row's table (this is often referred to as a *table scan*) or by preceding the row fetch with a lookup in a relevant index. The former approach is, needless to say, slow if only a small fraction of the rows are relevant for a query. Conversely, depending on which columns are indexed, the latter approach may be suboptimal if most of the rows are needed, e.g. in a query that summarizes a column value across all or most table rows. So, whether or not to utilize an index, and which index to use, depends on a number of factors (mostly about minimizing I/O), but luckily the database's query optimizer will make that decision for you - and will even do a good job in most cases.

The far most commonly used index type is a B-tree (B stands for "balanced"), and this is also the index type that will be created by default for all primary keys and when you execute a “create index” SQL statement.

In general, the B-tree index maps column values (called *keys*) to either pointers to table rows or the entire rows themselves. The B-tree's main advantage is that it is ordered and balanced. Due to the ordering we can quickly navigate to the relevant leaf node without visiting all nodes, and all leaf nodes are equally deep down the tree, making lookup times reliable, no matter what key we search for. In the database world we love all kinds of reliability!

The technical details of B-trees will not be covered here, but in case you are not familiar with them already, I really encourage you to read up on B-trees ASAP! Wikipedia or any introductory book on databases is a place to start.

Well, a few facts are too important to leave out:

Space

Each node in a B-tree is stored in a data page and may therefore induce an expensive I/O when accessed. New keys are inserted in the leaf pages, and whenever a page becomes full, it is split into two. And here is the important part: under normal circumstances 50% of the contents of the full page is put on the first node, and the rest on the second node. This gives us two nodes that are only half-way full, which is a good strategy if the keys are inserted in random order. However, filling the buffer with only half-full pages is clearly a sub-optimal utilization of the costly buffer space. If the keys being inserted are monotonically increasing, the database can utilize this fact to make a far more space efficient split strategy: SQL Server will fill leaf nodes entirely (splitting them 100% / 0% instead of 50% / 50%), and Oracle will even optimize splits of internal nodes, splitting them 90% / 10%. This optimization has a huge impact on the performance, as we shall see later.

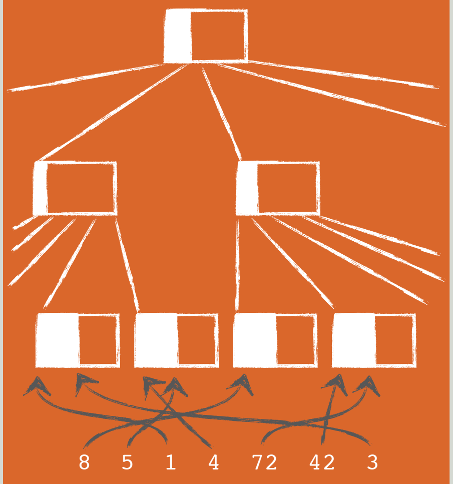
In the general case, we have no influence on the order in which keys are inserted, but in one important case we do: primary keys! So, using a monotonically increasing type as primary key will automatically imply far better space efficiency and thus a far better performance.

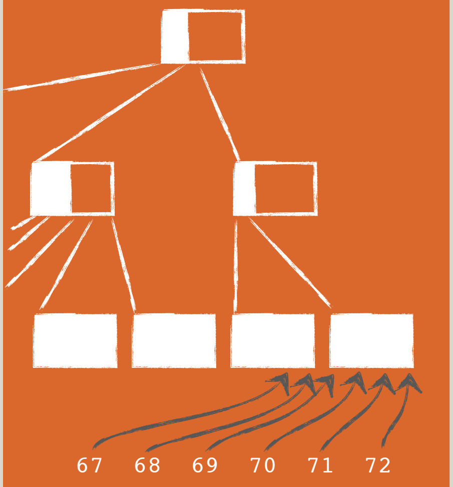
Order

There is one more reason for choosing a monotonically increasing type as primary key: a B-tree index can be a “clustering index” which means that the table rows are physically organized in the order given by the index key. In most, if not all, databases, a clustered index *is* the physical representation of the table, in the sense that the leaf pages contain the real data rows instead of just a pointer to them. A clustered index uniquely defines the row ordering, so at most one clustered index can exist per table. The order-by-key paradigm has some side effects: if keys for two rows are close, chances are high that these rows are located in the same page or in consecutive pages. In other words: when the keys somehow indicate a relationship between the rows, “related” (whatever this means) rows will tend to be on the same page or in consecutive pages, and they can therefore be accessed using a minimal amount of I/O. Conversely, if two keys are far apart, the corresponding rows will probably be stored in different pages, and, if not cached in buffer, require an increased amount of expensive I/O.

Let's have a closer look at three common choices of data types for primary keys: random GUIDs, sequential GUIDs and integers. Well, and strings...

KEY ORDER





Sequential key order is nice

In B-trees, keys are inserted at leaf level, and some of the keys propagate up in the tree to guide the search. If the keys are sequential, the leaf pages will be 100% filled, giving a really compact tree. Oracle even splits inner nodes cleverly in that case! If the key order is random, this optimization is not obtained.

Typical key types

Random GUIDs

A conventional GUID is a 16-byte Globally Unique IDentifier with a pseudo-random behavior which is guaranteed to be, well, world-wide unique. For the sake of abbreviation, let’s refer to them as *R-GUIDs*. Here is an example of three R-GUIDs:

```
0AF92AC9-8B45-4B9F-9E0D-E6A5C9023E91
8A9EF766-0A4E-437D-8DF9-E389CD1C7543
5B899EE7-7F0D-40FE-A788-4A20E8135E92
```

R-GUIDs are nice because

- ✓ they can be created programmatically without a roundtrip to the database. No single-point-of-origin, which is good when data is created in a distributed environment;
- ✓ R-GUIDs are unique across systems which is a practical property for data exchange;
- ✓ their random nature will spread consecutively inserted rows into different pages, thereby reducing or eliminating access contention on a single page;
- ✓ privacy concerns may require the IDs of related entities to be obfuscated and not be deducible from time-of-insertion.

But they also have some major drawbacks:

- their random nature will imply an awful lot of I/O for both reads and writes (SELECTs and INSERTs/UPDATEs);
- we have seen that choosing a primary key that somehow keeps related rows together will save I/O. But what is a good choice of relationship? It's certainly not random!
- in fact, because of their random nature, an R-GUID is probably the worst possible key for most practical use cases;
- 16 bytes are likely far more than needed. Especially for “normal” non-clustering indexes this will oversize the index significantly.

Sequential GUIDs

SQL Server provides a sequential GUID datatype (let's denote it *S-GUID*) that almost eliminates the largest drawback for R-GUIDs: the randomness. S-GUIDs may look like this:

```
21941C1C-6DAA-E011-AC65-6C626DCA5E45
23941C1C-6DAA-E011-AC65-6C626DCA5E45
25941C1C-6DAA-E011-AC65-6C626DCA5E45
```

Since S-GUIDs are created on the server, they cannot be created programmatically on the client side, but as shown later, this does not necessarily imply an increased number of database roundtrips. You should notice that S-GUIDs are not guaranteed to be sequential across a reboot. That is, the monotonically increasing sequence may get a new starting point when the server is rebooted. Advantages of S-GUIDs include:

- ✓ they are sequential and will therefore perform significantly better than R-GUIDs, as we shall see later;
- ✓ they are unique across systems;
- ✓ the datatype is still “GUID”, meaning that existing datatypes for table columns and client code can stay unaltered when transforming R-GUIDs to S-GUIDs.

Drawbacks of S-GUIDs:

- the strict monotonicity may break upon server reboots;
- there is a single-point-of-origin;
- they are still 16 bytes wide, which is a lot for an ID;
- the privacy property of R-GUIDs doesn't exist for S-GUIDs.

Integers

In the good old days, before the widespread use of GUIDs, a common choice for primary key was a sequentially increasing integer, which is a natively supported datatype in most databases. Integers are typically only 4 (or 8) bytes wide which is more than sufficient for most (all) cases. Think about it: will there ever pass more than 2^{31} = 2 billion rows (or the double amount for the unsigned type) through your table? Most likely not. And will there ever pass more than 2^{63} = 9,223[and 15 more digits] through? Certainly not! A signed 8 byte integer allows 29 million rows to be inserted every second for 10,000 years! Advantages of sequential integers:

- ✓ they are sequential and therefore have all the performance goodies that come with this property;
- ✓ the sequence is stable across reboots;
- ✓ they are only 4 bytes wide. Or 8 bytes in the very rare cases where 4 bytes is not enough. For unclustered indexes this gives a relatively better page utilization compared to GUIDs!
- ✓ they are easy to read and type in manually for testing purposes.

Drawbacks:

- an internal shared resource implies a small overhead for locking;
- the single-point-of-origin;
- the potential privacy problem described above.

Strings

Technically, there is also strings, which I have actually seen used as primary key type surprisingly often.

While they might be easy to read and interpret, they must either be entered manually or contain some auto-generated random or sequential element in which case a GUID or preferably an integer could, technically, have been used instead. The readability should (and can) be handled in other ways. The advantage of strings:

- ✓ they are easy to read.

Disadvantages:

- strings, and the data structures they imply, will most likely be overwhelmingly huge and space inefficient compared to a sequential integer;
- the potential privacy problem described above.

BUT...

1. In most cases, the single-point-of-origin property is not a problem: we do not have to use an extra database roundtrip to obtain the generated key as seen on the next page. And even if we did (which we don't), chances are that the performance gain obtained by switching from R-GUIDs to sequential integers far dominates the performance reduction induced by the extra roundtrip.
2. Personally, I have never experienced a situation where the order of row insertion had to be obfuscated by randomizing the primary key. Never. Such situations exist though, but be really really sure that your situation is one of them before you decide to use a random primary key.

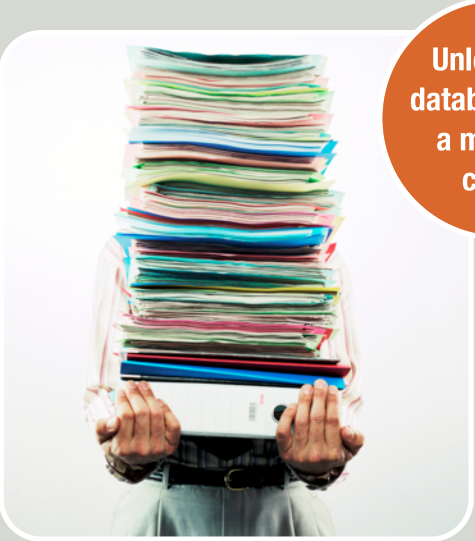
High-level memory caching

A high-level memory cache is not something the database provides natively, but nevertheless a concept everyone should consider as a fundamental part of their application's architecture: the database buffer is not buffering *high-level* objects in contrast to a memory cache which stores the structures *exactly* as we need them, thereby saving (expensive) build time; the overhead from the database access may constitute a significant part of the total transaction time; and by using a *distributed* cache, the amount of memory practically becomes unlimited. So, if an application uses a number of data structures that require more than a moderate amount of database or CPU work to build, a significant performance boost can probably be achieved by caching these high-level data structures in a fast memory cache.

The expected impact from a memory cache depends on the data structure access pattern, but generally increases with the ratio between how often they are read and updated. Actually, I will make a somewhat bold statement: in the general case, if you ever expect your application to scale well, a high-level object memory cache is simply a must!

The great thing about caches is that they can be implemented iteratively object-by-object, starting with the heaviest bottleneck-object first.

A common choice of memory cache that is also used by giants such as Wikipedia, Flickr, Twitter, YouTube, and others is *memcached*, which is an extremely (!) fast and distributed memory cache. In short, *distributed* means that if you ever need more cache memory, just add machines to the cache pool, and you are good to go.



Unload the database with a memory cache

RANDOM FACT #1

In SQL Server, sequential integers and S-GUIDs can be generated using, respectively, the SQL:

```
create table myIntTable(
    id int not null identity primary key,
    stuff varchar(100)
);

insert into myIntTable(stuff) values ('someStuff');
```

and

```
create table mySGUIDTable(
    id uniqueidentifier not null
       default newsequentialid() primary key,
    stuff varchar(100)
);

insert into mySGUIDTable(stuff) values ('someStuff');
```

...

We can retrieve the generated IDs by using the "output clause" in TSQL, or, for sequential integers, by using two build-in functions. No need for an extra database roundtrip:

scope_identity() returns the latest generated identity across all tables within the current connection.

ident_current('tableName') returns the latest generated identity for the specified table within the current connection.

Example:

```
   insert into myIntTable(stuff) values ('someStuff');
   insert into myChildTable(parentId, otherStuff)
       values (scope_identity(), 'someOtherStuff');
```

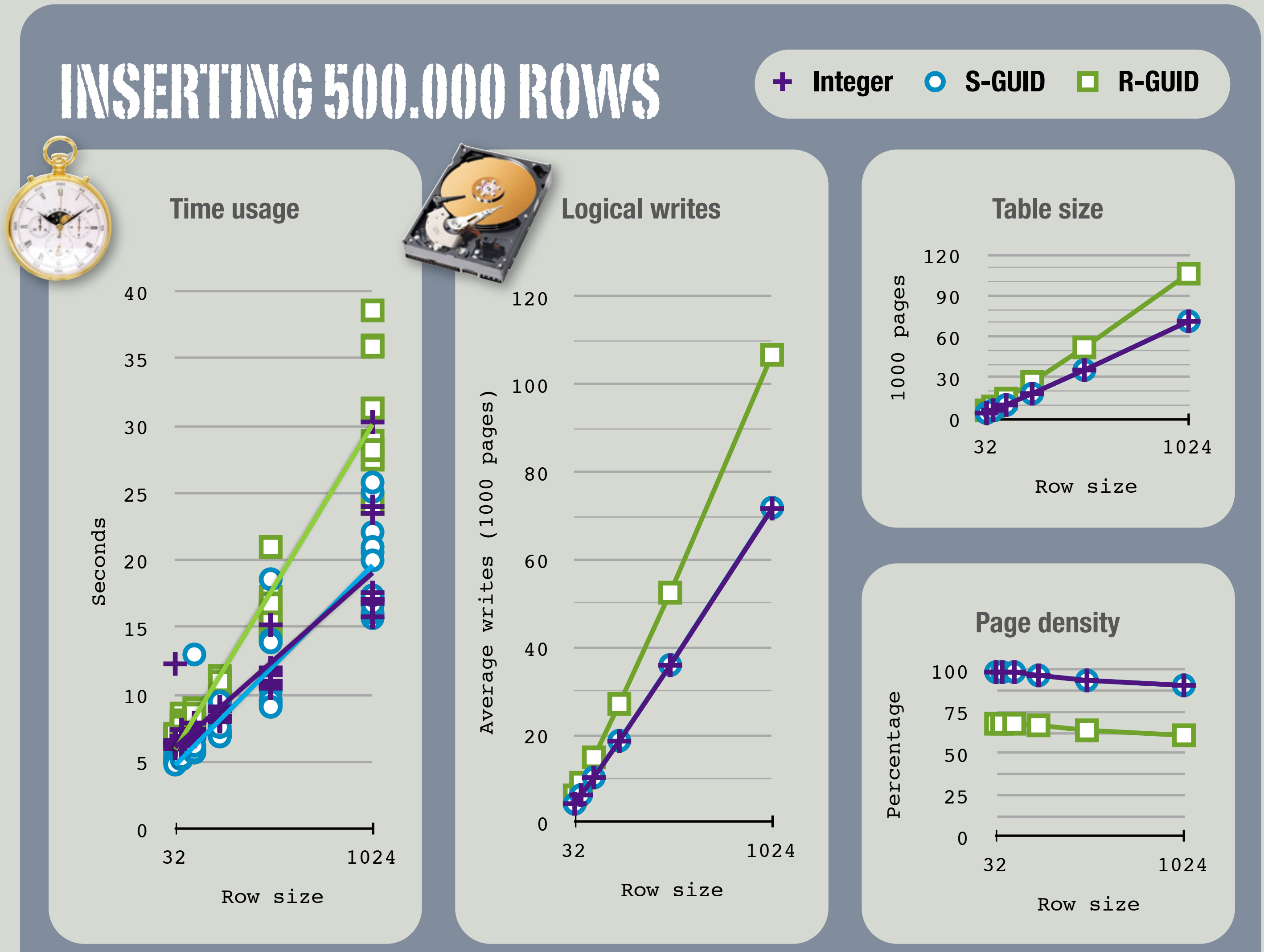
RANDOM FACT #2

Implementing a memory cache can be easy

```
GetProduct(id) {
   cacheKey = 'products' + id
   if(ExistsInCache(cacheKey))
      return GetFromCache(cacheKey);
   product = BuildComplicatedProduct(id);
   StoreInCache(cacheKey, product);
   return product;
}

StoreProduct(product)
{
   UpdateDatabase(product);
   cacheKey = 'products' + product.id;
   RemoveFromCache(cacheKey);
}
```

INSERTING 500.000 ROWS



The graphs compare performance of row insertion for varying row size and type of primary key. Shorter rows implies a smaller amount of I/O and a smaller execution time. Also, R-GUIDs imply a lower page density, a larger table, more I/O and thus poorer performance.

Some practical experiments

A number of practical experiments was carried out to see if the theory from the preceding pages really holds. The experiments were performed on a 2.8 GHz 8 core machine with 8 GB RAM, and a 7200 RPM SATA disk, running 64 bit Windows 7 and SQL Server 2008. We compare the efficiency of three basic classes of operations for varying type of primary key, and varying row size. The operations measured are:

1. insertion of 500.000 new rows in a table;
2. read access to 100 rows in the order of their insertion;
3. read access to 100 randomly picked rows.

The sequence of experiments was executed 10 times in order to minimize the influence of unrelated events on the machine, and the measurements shown here are (mostly) the averages of the seen numbers.

The row sizes shown in the graphs are the total column widths. Note that the shown sizes exclude the metadata associated to each row (at least 9 bytes).

Results for row insertion

The graphs in the box above illustrate the results for row insertion. As the variance in execution time was relatively large for this experiment, the first graph shows the execution time for each individual test run together with a linear trend line. There are three immediate observations:

1. the execution time increases with the row width;
2. the execution time is almost given by the amount of I/O; and
3. integers and S-GUIDs perform almost equally good, while R-GUIDs perform poorly.

The two smaller graphs to the right give a good explanation to these observations: as the rows grow, so does the total table size, and our buffer hit ratio consequently decreases. Also, the pages storing the table with an R-GUID as primary key is packed much less efficiently (less than 75% full) than the pages holding integer and S-GUID keys (which are almost 100% full).

Quick bonus info: a call to identity() for generating a sequential integer is actually significantly slower than a call to newsequentialid() for generating an S-GUID. This is because the former needs to lock a shared resource, while the latter generates the ID without maintaining an internal state.

Results for row reading

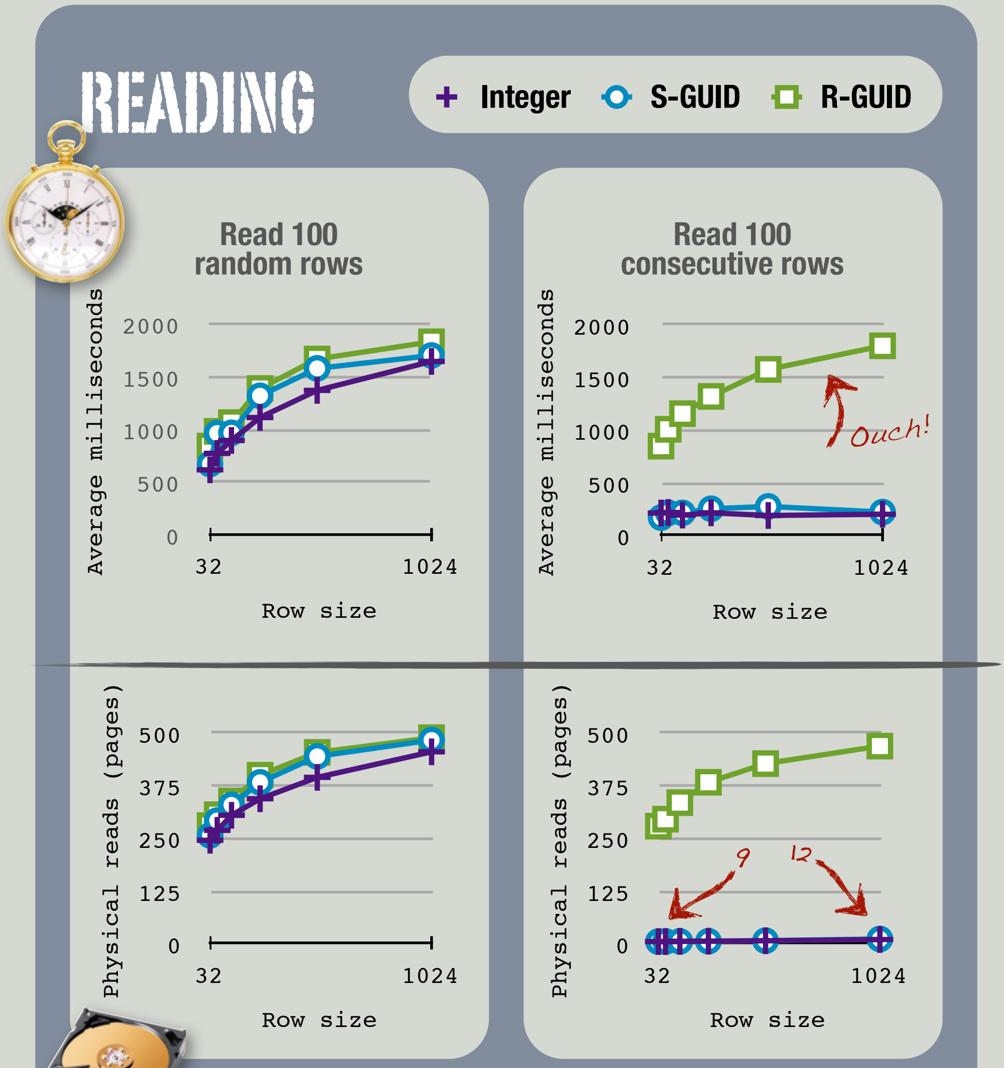
Take a look at the graphs to the right, illustrating the execution time and I/O for reading 100 rows in random and sequential order, respectively. In the first case, reading random rows, all three datatypes perform almost identically, with a small advantage to the integers. As we shall see in a moment, the performance for all datatypes is poor, and this is because random access is the most buffer-unfriendly access pattern imaginable.

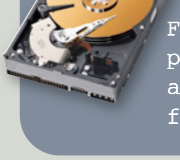
In contrast, when reading consecutive or nearly-consecutive rows, we can really take advantage of the buffer: many of the rows will exist in the same already buffered page, and by prefetching multiple pages, the total amount of I/O costs can also be optimized. This can be seen in the rightmost graphs, where integers and S-GUIDs totally outperform R-GUIDs with a minimum amount of I/O, and an almost non-increasing execution time when the row size increases. In these experiments, integers and S-GUIDs are 4 to 10 times faster than R-GUIDs.

Unclustered indexes

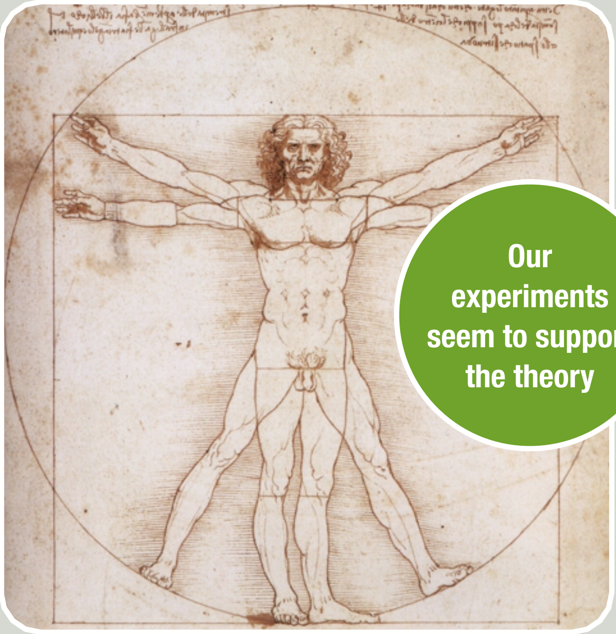
Not surprisingly, we see that an unclustered index on a 4 byte integer is smaller than an index on any 16 byte GUID (see graph). And as for tables and clustered indexes, wider index keys means larger index structure. Notice that the GUID index is actually less than 4 times larger than the integer index. This is because indexes also store metadata with each entry.

READING



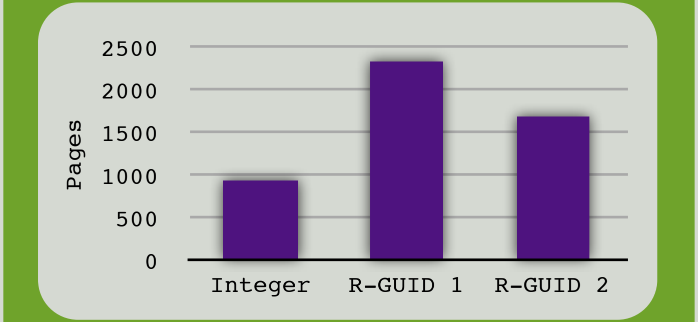


For random access, the execution time suffers from poor buffer and I/O utilization. For sequential access, the buffer really speeds things up! Well, for sequential keys (not R-GUIDs), that is...



Our experiments seem to support the theory

UNCLUSTERED INDEX



An unclustered index on 500,000 values is smaller for integers than for R-GUIDs. The "R-GUID 1" bar above shows the index size immediately after insertion, while "R-GUID 2" shows the size after an index re-build.

LESSONS LEARNED

Basically, there is only one fundamental goal: save space, save I/O! This implies...

Great design, great speed

- **R-GUIDs perform horribly as primary keys** as soon as the number of rows is anything but trivial. If no other choice is obvious, use (small) sequential integers instead as the database can utilize their monotonicity to highly improve both space utilization and I/O costs. Well, as described earlier, there exist situations where random keys are exactly what you want, but these situations are rare, and you should be absolutely certain about your privacy or contention problem before following that path. To read more about this and relevant situations, google “reverse key indexing”.
- **Use small datatypes** whenever possible. Every bit counts, so never use a larger datatype than needed. Small datatypes give a better space utilization and will likely give you a higher buffer hit ratio. This is especially true for unclustered indexes where a change of column width will be higher, relative to the total row width in the index.
- Talking about space: be careful to **only include relevant columns in multi-column indexes**. Needless to say, the more columns, the wider index entries.
- Be careful when representing various kinds of **categories and semi-IDs as strings**. Especially when the number of rows is large. Strings are wider than integers, so probably you would get a better performance by putting these strings/labels in a separate ID-to-label table and referring to the relevant (small integer) IDs in the large original table (and in your code) instead. If the amount of categories/labels is small, the ID-to-label table will quickly exist entirely in buffer. Besides, you can then easily rename a category without destroying the references to it - just rename the relevant row in the ID-to-label table.
- A common way to reduce the need for table joins is to replicate columns from one of the involved tables into the other table(s). This technique is called *de-normalization*. Surely, this may avoid accessing the table whose columns are now replicated, but it also increases the row size of the target table. Furthermore, all copies of the replicated columns must be accessed when updated! Depending on the access pattern, the performance decrease induced by the lower buffer hit ratio and increased write costs may be much larger than the performance increase gained from saving the join. In the general case you should therefore **avoid de-normalization**, and be absolutely sure that it will actually have a positive performance impact before you do it.
- **Clustering rows by a foreign key** is a way to co-locate related rows. However, carefully analyze that the actual access pattern benefits from this choice of clustering (pay special attention to the page density and sequential access of subsets within a single foreign key).
- If your tables are defined indirectly from code (programmatically) using a class persister, be careful to **only persist clean native types** as table columns: depending on the class persister, a high-level class defined as “column” may be expanded to multiple native-type columns by the class persister. This can be an expensive affair due to the possible considerable extra row width. There is probably a reason why the high-level class is defined as its own, instead of being “merged” with the class you are persisting: if there is a one-to-many or many-to-many relationship you will get multiple identical occurrences of the same object in your table. Not space efficient. And if you modify one of the high-level objects, all its representatives in the table will have to be updated as well, probably causing a massive amount of I/O. Chances are that you would be better of with a *normalized* design, putting the high-level objects in their own table and referring to them by their primary key (e.g. a small integer) in the main table instead.
- **Use a high-level memory cache**, such as memcached, to speed up the retrieval of high-level data structures by taking off the corresponding load from the database and CPU. When used right, a memory cache can be the difference between snail and rail!

Further reading

There has been written a lot on database tuning, and it can be hard to find something good. I recommend the following:

- A great all-round book that covers a lot of tuning tips, loaded with practical experiments: *Database Tuning - Principles, Experiments, and Troubleshooting Techniques* by Dennis Elliott Shasha, Philippe Bonnet and Jim Grey.
- An inspiring website on high scalability with lots of insights from the big players and with numerous suggestions for further reading: www.highscalability.com
- The “behind the scenes” book for your database. E.g. *Microsoft SQL Server 2008 Internals* by Kalen Delaney.
- The blogs on sqlskills.com (for SQL Server) or asktom.oracle.com (for Oracle).

Check it out...

Did you know that databases are not all “row stores” like SQL Server, Oracle, DB2, MySQL, PostgrSQL, etc? Other types may better fit your needs. Google the following keywords: “cassandra”, “vectorwise”, “voltdb.com”, “vertica.com”, “column store”, “key value store”, “hadapt”.