When and why are database joins expensive?

Asked 16 years, 2 months ago Modified 3 months ago

Viewed 108k times

419

I'm doing some research into databases and I'm looking at some limitations of relational DBs.

I'm getting that joins of large tables is very expensive, but I'm not completely sure why. What does the DBMS need to do to execute a join operation, where is the bottleneck? How can denormalization help to overcome this expense? How do other optimization techniques (indexing, for example) help?

Personal experiences are welcome! If you're going to post links to resources, please avoid Wikipedia. I know where to find that already.

In relation to this, I'm wondering about the denormalized approaches used by cloud service databases like BigTable and SimpleDB. See this question.

database performance join relational-database denormalization

Share

edited May 23, 2017 at 12:34

Improve this question

Follow

Community Bot

1 ● 1

asked Oct 6, 2008 at 9:52

Rik

29.2k ● 14 ● 51 ● 69

1 I'm am looking into an objective (if there is such a thing) comparison. Pro's, con's, what-have-you's. – Rik Oct 6, 2008 at 13:16

1 This question is too broad because of the significant differences in the implementation of JOINing among DB vendors. – Rick James Aug 18, 2022 at 18:15

8 Answers

Sorted by: Highest score (default)

548

Denormalising to improve performance? It sounds convincing, but it doesn't hold water.

Chris Date, who in company with Dr Ted Codd was the original proponent of the relational data model, ran out of patience with misinformed arguments against normalisation and systematically demolished them using scientific method: he got large databases and *tested* these assertions.

I think he wrote it up in *Relational Database Writings 1988-1991* but this book was later rolled into edition six of *Introduction to Database Systems*, which is *the* definitive text on database theory and design, in its eighth edition as I write and likely to remain in print for decades to

come. Chris Date was an expert in this field when most of us were still running around barefoot.

He found that:

- Some of them hold for special cases
- All of them fail to pay off for general use
- All of them are significantly worse for other special cases

It all comes back to mitigating the size of the working set. Joins involving properly selected keys with correctly set up indexes are cheap, not expensive, because they allow significant pruning of the result *before* the rows are materialised.

Materialising the result involves bulk disk reads which are the most expensive aspect of the exercise by an order of magnitude. Performing a join, by contrast, logically requires retrieval of only the *keys*. In practice, not even the key values are fetched: the key hash values are used for join comparisons, mitigating the cost of multi-column joins and radically reducing the cost of joins involving string comparisons. Not only will vastly more fit in cache, there's a lot less disk reading to do.

Moreover, a good optimiser will choose the most restrictive condition and apply it before it performs a join, very effectively leveraging the high selectivity of joins on indexes with high cardinality.

Admittedly this type of optimisation can also be applied to denormalised databases, but the sort of people who *want* to denormalise a schema typically don't think about cardinality when (if) they set up indexes.

It is important to understand that table scans (examination of every row in a table in the course of producing a join) are rare in practice. A query optimiser will choose a table scan only when one or more of the following holds.

- There are fewer than 200 rows in the relation (in this case a scan will be cheaper)
- There are no suitable indexes on the join columns (if it's meaningful to join on these columns then why aren't they indexed? fix it)
- A type coercion is required before the columns can be compared (WTF?! fix it or go home) **SEE END NOTES FOR ADO.NET ISSUE**
- One of the arguments of the comparison is an expression (no index)

Performing an operation is more expensive than not performing it. However, performing the *wrong* operation, being forced into pointless disk I/O and then discarding the dross prior to performing the join you really need, is *much* more expensive. Even when the "wrong" operation is precomputed and indexes have been sensibly applied, there remains significant penalty. Denormalising to precompute a join - notwithstanding the update

anomalies entailed - is a commitment to a particular join. If you need a *different* join, that commitment is going to cost you *big*.

If anyone wants to remind me that it's a changing world, I think you'll find that bigger datasets on gruntier hardware just exaggerates the spread of Date's findings.

For all of you who work on billing systems or junk mail generators (shame on you) and are indignantly setting hand to keyboard to tell me that you know for a fact that denormalisation is faster, sorry but you're living in one of the special cases - specifically, the case where you process *all* of the data, in-order. It's not a general case, and you *are* justified in your strategy.

You are *not* justified in falsely generalising it. See the end of the notes section for more information on appropriate use of denormalisation in data warehousing scenarios.

I'd also like to respond to

> Joins are just cartesian products with some lipgloss

What a load of bollocks. Restrictions are applied as early as possible, most restrictive first. You've read the theory, but you haven't understood it. Joins are *treated* as "cartesian products to which predicates apply" *only* by the query optimiser. This is a symbolic representation (a normalisation, in fact) to facilitate symbolic decomposition

so the optimiser can produce all the equivalent transformations and rank them by cost and selectivity so that it can select the best query plan.

The only way you will ever get the optimiser to produce a cartesian product is to fail to supply a predicate: `SELECT * FROM A,B`

Notes

David Aldridge provides some important additional information.

There is indeed a variety of other strategies besides indexes and table scans, and a modern optimiser will cost them all before producing an execution plan.

A practical piece of advice: if it can be used as a foreign key then index it, so that an index strategy is *available* to the optimiser.

I used to be smarter than the MSSQL optimiser. That changed two versions ago. Now it generally teaches *me*. It is, in a very real sense, an expert system, codifying all the wisdom of many very clever people in a domain sufficiently closed that a rule-based system is effective.

"Bollocks" may have been tactless. I am asked to be less haughty and reminded that math doesn't lie. This is true, but not all of the implications of mathematical models

should necessarily be taken literally. Square roots of negative numbers are very handy if you carefully avoid examining their absurdity (pun there) and make damn sure you cancel them all out before you try to interpret your equation.

The reason that I responded so savagely was that the statement as worded says that

> Joins *are* cartesian products...

This may not be what was meant but it *is* what was written, and it's categorically untrue. A cartesian product is a relation. A join is a function. More specifically, a join is a relation-valued function. With an empty predicate it will produce a cartesian product, and checking that it does so is one correctness check for a database query engine, but nobody writes unconstrained joins in practice because they have no practical value outside a classroom.

I called this out because I don't want readers falling into the ancient trap of confusing the model with the thing modelled. A model is an approximation, deliberately simplified for convenient manipulation.

The cut-off for selection of a table-scan join strategy may vary between database engines. It is affected by a number of implementation decisions such as tree-node fill-factor, key-value size and subtleties of algorithm, but

broadly speaking high-performance indexing has an execution time of $k \log n + c$. The C term is a fixed overhead mostly made of setup time, and the shape of the curve means you don't get a payoff (compared to a linear search) until n is in the hundreds.

Sometimes denormalisation is a good idea

Denormalisation is a commitment to a particular join strategy. As mentioned earlier, this interferes with *other* join strategies. But if you have buckets of disk space, predictable patterns of access, and a tendency to process much or all of it, then precomputing a join can be very worthwhile.

You can also figure out the access paths your operation typically uses and precompute all the joins for those access paths. This is the premise behind data warehouses, or at least it is when they're built by people who know why they're doing what they're doing, and not just for the sake of buzzword compliance.

A properly designed data warehouse is produced periodically by a bulk transformation out of a normalised transaction processing system. This separation of the operations and reporting databases has the very desirable effect of eliminating the clash between OLTP and OLAP (online transaction processing ie data entry, and online analytical processing ie reporting).

An important point here is that apart from the periodic updates, the data warehouse is *read only*. This renders moot the question of update anomalies.

Don't make the mistake of denormalising your OLTP database (the database on which data entry happens). It might be faster for billing runs but if you do that you will get update anomalies. Ever tried to get Reader's Digest to stop sending you stuff?

Disk space is cheap these days, so knock yourself out. But denormalising is only part of the story for data warehouses. Much bigger performance gains are derived from precomputed rolled-up values: monthly totals, that sort of thing. It's *always* about reducing the working set.

ADO.NET problem with type mismatches

Suppose you have a SQL Server table containing an indexed column of type varchar, and you use AddWithValue to pass a parameter constraining a query on this column. C# strings are Unicode, so the inferred parameter type will be NVARCHAR, which doesn't match VARCHAR.

VARCHAR to NVARCHAR is a widening conversion so it happens implicitly - but say goodbye to indexing, and good luck working out why.

"Count the disk hits" (Rick James)

If everything is cached in RAM, `JOINs` are rather cheap. That is, normalization does not have much *performance penalty*.

If a "normalized" schema causes `JOINs` to hit the disk a lot, but the equivalent "denormalized" schema would not have to hit the disk, then denormalization wins a performance competition.

> Comment from original author: Modern database engines are very good at organising access sequencing to minimise cache misses during join operations. The above, while true, might be miscontrued as implying that joins are necessarily problematically expensive on large data. This would lead to cause poor decision-making on the part of inexperienced developers.

edited Jan 28, 2016 at 3:17

community wiki
20 revs, 6 users 88%
Peter Wone

7 Sonme of these statements are specific to a particular DBMS, aren't they? eg. "There are fewer than 200 rows in the relation" – David Aldridge Oct 6, 2008 at 12:52

In the general case the '200' rows boils down to 'how much table data fits into a single disk operation'. The DB will read data in lots of one or more disk blocks (on Oracle, for example, this is configurable). This is one disk op. Reading from and index and then the table is two or more ops. – ConcernedOfTunbridgeWells Oct 11, 2008 at 22:41

Table scans can also be chosen when you're trying to seek more than x% of rows. I've heard for Sql Server, that x is anywhere from 10-20%. – Mark Brackett Nov 3, 2008 at 23:40

2 Does the use of surrogate keys (or not) influence all this significantly? – David Plumpton May 5, 2009 at 22:46

1 I'd add "You're using MySQL" to your list of special cases, as it's query optimiser is piss poor to the point where denormalising is sometimes the only viable choice (though why use mysql, other than ignorance?). – GordonM Jan 5, 2015 at 15:57

53

What most commenters fail to note is the wide range of join methodologies available in a complex RDBMS, and the denormalisers invariably gloss over the higher cost of maintaining denormalised data. Not every join is based on indexes, and databases have a lot of optimised algotithms and methodologies for joining that are intended to reduce join costs.

In any case, the cost of a join depends on its type and a few other factors. It needn't be expensive at all - some examples.

- A hash join, in which bulk data is equijoined, is very cheap indeed, and the cost only become significant if the hash table cannot be cached in memory. No index required. Equi-partitioning between the joined data sets can be a great help.
- The cost of a sort-merge join is driven by the cost of the sort rather than the merge -- an index-based access method can virtually eliminate the cost of the sort.
- The cost of a nested loop join on an index is driven by the height of the b-tree index and the access of the table block itself. It's fast, but not suitable for bulk joins.
- A nested loop join based on a cluster is much cheaper, with fewer logicAL IO'S required per join row -- if the joined tables are both in the same cluster then the join becomes very cheap through the colocation of joined rows.

Databases are designed to join, and they're very flexible in how they do it and generally very performant unless they get the join mechanism wrong.

edited Oct 6, 2008 at 13:00

answered Oct 6, 2008 at 12:48

David Aldridge
52.3k ● 8 ● 72 ● 99

I think it comes down to "if in doubt, ask your DBA". Modern databases are complex beasts and do require study to understand. I've only been using Oracle since 1996 and it's a full time job keeping up with the new features. SQLserver has also come along hugely way since 2005. It's not a black box! – Guy Oct 8, 2008 at 2:48

3 Hmmm, well in my humble experience there are too many DBA's out there who have never heard of a hash join, or think that they're a Universally Bad Thing. – David Aldridge Oct 8, 2008 at 16:15

39

I think the whole question is based on a false premise. Joins on large tables are *not* necessarily expensive. In fact, **doing joins efficiently is one of the main reasons relational databases exist** at all. Joins on large *sets* often are expensive, but very rarely do you want to join the entire contents of large table A with the entire contents of large table B. Instead, you write the query such that *only the important rows* of each table are used and the actual set kept by the join remains smaller.

Additionally, you have the efficiencies mentioned by Peter Wone, such that only the important parts of each record need be in memory until the final result set is materialized. Also, in large queries with many joins you typically want to start with the smaller table sets and work your way up to the large ones, so that the set kept in memory remains as small as possible as long as possible.

When done properly, joins are generally the *best way* to compare, combine, or filter on large amounts of data.

Share Improve this answer Follow

edited Oct 6, 2008 at 16:45

answered Oct 6, 2008 at 13:53

Joel Coehoorn

415k ● 114 ● 577 ● 813

1 @joel. The converse is also true. Large dataset joins can be expensive and are sometimes required, but you don't want to do it too often unless a) you can handle the IO and RAM needed and b) you're not doing it too often. Consider materialised views, reporting systems, realtime vs CoB reports. – Guy Oct 8, 2008 at 2:43

16

The bottleneck is pretty much *always* disk I/O, and even more specifically - random disk I/O (by comparison, sequential reads are fairly fast and can be cached with read ahead strategies).

Joins *can* increase random seeks - if you're jumping around reading small parts of a large table. But, query optimizers look for that and will turn it into a sequential table scan (discarding the unneeded rows) if it thinks that'd be better.

A single denormalized table has a similar problem - the rows are large, and so less fit on a single data page. If you need rows that are located far from another (and the

large row size makes them further apart) then you'll have more random I/O. Again, a table scan may be forced to avoid this. But, this time, your table scan has to read more data because of the large row size. Add to that the fact that you're *copying data* from a single location to multiple locations, and the RDBMS has that much more to read (and cache).

With 2 tables, you also get 2 clustered indexes - and can generally index more (because of less insert/update overhead) which can get you drastically increased performance (mainly, again, because indexes are (relatively) small, quick to read off disk (or cheap to cache), and lessen the amount of table rows you need to read from disk).

About the only overhead with a join comes from figuring out the matching rows. Sql Server uses 3 different types of joins, mainly based on dataset sizes, to find matching rows. If the optimizer picks the wrong join type (due to inaccurate statistics, inadequate indexes, or just an optimizer bug or edge case) it can drastically affect query times.

- A loop join is farily cheap for (at least 1) small dataset.
- A merge join requires a sort of both datasets first. If you join on an indexed column, though, then the index is already sorted and no further work needs to be done. Otherwise, there is some CPU and memory overhead in sorting.

- The hash join requires both memory (to store the hashtable) and CPU (to build the hash). Again, this is fairly quick in relation to the disk I/O. *However*, if there's not enough RAM to store the hashtable, Sql Server will use tempdb to store parts of the hashtable and the found rows, and then process only parts of the hashtable at a time. As with all things disk, this is fairly slow.

In the optimal case, these cause no disk I/O - and so are negligible from a performance perspective.

All in all, at worst - it should actually be faster to read the same amount of *logical* data from x joined tables, as it is from a single denormalized table because of the smaller disk reads. To read the same amount of *physical* data, there could be some slight overhead.

Since query time is usually dominated by I/O costs, and the size of your data does not change (minus some very miniscule row overhead) with denormalization, there's not a tremendous amount of benefit to be had by just merging tables together. The type of denormalization that tends to increase performance, IME, is caching calculated values instead of reading the 10,000 rows required to calculate them.

answered Nov 3, 2008 at 23:33

Mark Brackett

85.6k ● 17 ● 111 ● 155

Reducing random seeks: good point, although a good RAID controller with a big cache will do elevator read/write.
– Peter Wone Mar 19, 2009 at 12:00

The best answer in the thread! Covered the most significant aspects and their effect on disk, CPU and RAM. Though, the conclusion about denormalisation is valid only for reading large data. Modern apps usually handle paginated requests with modest output. In this case denormalisation wins.
– Alex Klaus Nov 10, 2020 at 11:52

3

The order in which you're joining the tables is extremely important. If you have two sets of data try to build the query in a way so the smallest will be used first to reduce the amount of data the query has to work on.

For some databases it does not matter, for example MS SQL does know the proper join order most of the time. For some (like IBM Informix) the order makes all the difference.

Share Improve this answer

Follow

answered Oct 6, 2008 at 9:58

Ilya Kochetov
18.4k ● 6 ● 47 ● 62

1 In general a decent query optimizer is going to be unaffected by the order that the joins or tables are listed, and will make its own determination of the most efficient way to perform the join. – David Aldridge Oct 6, 2008 at 22:56

7 MySQL, Oracle, SQL Server, Sybase, postgreSQL,etc. care not the order of joins. I've worked with DB2 and it also, to my knowledge, doesn't care what order you put them in. This is

not helpful advice in the general case – Matt Rogish Oct 7, 2008 at 20:24

MySQL clustering using the NDB engine (admittedly an edge case, and only advanced developers are going to go near NDB) doesn't guess the join order correctly, so you have to add "USE INDEX" statements to most joined queries or they'll be horrendously inefficient. MySQL docs cover it. – joelhardi Oct 8, 2008 at 2:16

@iiya, Understanding what the optimiser will choose is more important than generalised statements or "myths" about table ordering. Do not rely on a particular quirk in your SQL as the behaviour often changes when the RDBMS is upgraded. Oracle has changed behaviours several times since v7. – Guy Oct 8, 2008 at 2:37

1 @Matt I've saw Oracle 9i perform very different optimizations and query plans just adjusting the join order. Maybe this has changed from version 10i onwards? – Camilo Díaz Repka Feb 28, 2009 at 17:34

0

Deciding on whether to denormalize or normalize is fairly a straightforward process when you consider the complexity class of the join. For instance, I tend to design my databases with normalization when the queries are O(k log n) where k is relative to the desired output magnitude.

An easy way to denormalize and optimize performance is to think about how changes to your normalize structure affect your denormalized structure. It can be problematic however as it may require transactional logic to work on a denormalized structured.

The debate for normalization and denormalization isn't going to end since the problems are vast. There are many problems where the natural solution requires both approaches.

As a general rule, I've always stored a normalized structure and denormalized caches that can be reconstructed. Eventually, these caches save my ass to solve the future normalization problems.

Share Improve this answer

Follow

answered Sep 20, 2009 at 12:25

MathGladiator

1,211 ● 1 ● 10 ● 24

0

A point of view of real-world physics:

My background is an app developer, not a DBA.

After indexing, data locality is the major factor when it comes to reading and writing data. On disk, reads can be slower than writes (think latency, not throughput). This can be mitigated with caches, eventually consistent replication etc. but I am speaking here purely of the database as the primary source of truth.

Whatever mechanism you use to store data, localizing all your related data in one place can provide a huge speed-up in read-heavy environments (eg when you are reading rows 100x more often than writing them).

This is true whether we're talking about sectors on a spinning disk or RAID array, or whether we are talking about shards, or even memory-mapped pages in RAM (in the latter case, the CPU often has an L2 cache that will grab data nearby).

So, modern RDBMSs support a variable-length rows (unless they're column-based databases). The indexes (indices?) point you to the location of the row, including the disk sector etc. Then an I/O process grabs the data. It is *this process* that can be slowed down if the data is all over the disk.

The same goes for shards. You should localize your data with the same primary keys on the same shards. For queries utilizing merely unique keys, you might have to fan out and search ALL shards. For relations between tables, I tend to create TWO intermediate tables, one for each direction of the relation, so that they can be sharded easily (I tend to discourage queries on non-primary indexes, for sharding reasons).

If you are *really* constantly reading all over your database all the time, none of this data locality stuff matters because the disk is spinning and the head is going all over it and the queries are coming in from all over the disk anyway. And same with shards. But this goes to your schema and data access patterns:

Do you really need to have your entire database available to be read from at any time? Can't you archive older posts, say, in slower and less replicated storage?

Colocate the data that everyone uses frequently in the same database tables on the same storage medium, and then have many caches / replicas only of the hot data.

That last part is the real reason why this thing doesn't really matter for common requests, and why data locality (i.e. denormalization) gives you a slight win in *latency*. Because if you set up caching layers (eg a CDN that caches your front page for a minute) generic common requests (eg for your front page) rarely even hit your app, let alone the database.

If you keep your free site Web 1.0 (only a few writers), or even static Web 2.0 — meaning very little difference regardless of who logged in — then your data access pattern and latency is irrelevant because everyone will only ever see a slightly-out-of-date cached version.

Economics of storage

The parts of your site that are heavily custom (eg personalized inbox or schedule of events) are the only places where any of this matters. For those, you can charge people a recurring membership fee if their resource use gets out of hand. You can also charge them for posting each new piece of content to your site, that you'll have to store and host etc. You can also charge them a micropayment for every time someone watches their content, and transfer the credits to the one who watched. But you can also allow people to upvote their content thereby transferring credits back, so posting

content can be an investment and this incentivizes content that people are more likely to upvote with their scarce credits.

Summary

Data locality improves latency. For most static pages / views on an app, this is irrelevant due to caching. It only ever matters for dynamic views (eg an inbox). For those views, you are likely going to have to deal more with storage costs, and I have explained how to pass them on to your users (producing and consuming content) in order to incentivize better quality content on your network.

Share Improve this answer

Follow

edited Aug 27 at 0:35

philipxy

15.1k ● 6 ● 42 ● 94

answered Aug 4 at 16:31

Gregory Magarshak

2,019 ● 2 ● 26 ● 38

-8

Elaborating what others have said,

Joins are just cartesian products with some lipgloss. {1,2,3,4}X{1,2,3} would give us 12 combinations (nXn=n^2). This computed set acts as a reference on which conditions are applied. The DBMS applies the conditions (like where both left and right are 2 or 3) to give us the matching condition(s). Actually it is more optimised but the problem is the same. The changes to

size of the sets would increase the result size exponentially. The amount of memory and cpu cycles consumed all are effected in exponential terms.

When we denormalise, we avoid this computation altogether, think of having a colored sticky, attached to every page of your book. You can infer the information with out using a reference. The penalty we pay is that we are compromising the essence of DBMS (optimal organisation of data)

Share Improve this answer

Follow

answered Oct 6, 2008 at 11:09

questzen

3,287 ● 20 ● 21

4 -1: This post is a great example of why you let the DBMS perform the joins -- because the DBMS designers think about these issues all the time and come up with more effective ways to do it than the compsci 101 method. – David Aldridge Oct 6, 2008 at 12:57

2 This answer is incorrect. If your query is executed against a normalized, indexed database and has any kind of filter or join condition, the optimizer will find a way to avoid the Cartesian product and minimize memory usage and CPU cycles. If you actually intend to select a Cartesian product, you'll use the same memory in a normalized or de-normalized db. – rileymcdowell Jan 29, 2019 at 21:37

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.