Jacques Nadeau: Hey, everybody, thanks for being here today. I am going to do a talk on … Well, it's a long title, "Using Apache Arrow, Calcite, Parquet, all to build a Relational Cache." This is sort of our experience at Dremio, what we did and talking about how these different technologies work together to solve performance problems and get to your data more quickly.

Quickly, who am I? I am the CTO and co-founder of Dremio, very active in a number of open source communities. I'm an Apache member, VP of Apache Arrow which is new project that's been around for about a year and a half. I'm also a PMC of Arrow, Calcite, the Apache Incubator as well as Heron which is an incubating project that came out of Twitter for streaming purposes.

Quickly, agenda … This doesn't move so I'll stay real close here. Quickly, agenda, I'm going to do a quick tech backgrounder on each of the three technologies that I'm talking about, and then I'm going to go into talking about different kinds of caching techniques. And specifically relational caching, go into depth on how that works, what we do when we're trying to do definition of relational caching as well as matching any relational caching and then talk a little about, as you need to with caching, is updates and how you deal with changing data and then a few closing words. I'm go through this pretty quick but afterwards, I have office hours. If you want to drill into a particular topic in more detail, then I would be happy to do so but I'm going to try to cover a wide range of things to get the idea across here.

Let's start out with the tech backgrounders. What are the three projects we're talking about? The first is Apache Arrow. Apache Arrow, as I mentioned, is about a year and a half old. It is a project focused on in-memory columnar processing specifically for various types of big data purposes. It has two main components to it. The first is a common representation of data in memory no matter what type of language you're using, and the second is a collection of different algorithms and libraries to work with this representation of data. Designed to work with any programming language, and it's focused on both what you think of traditional relational data which is rows and columns data as well as more common, more modern data like complex data, nested data, heterogeneous types, that kind of thing.

Arrow, even though it's fairly new, has been incorporated in some really interesting projects. One is that Pandas now is incorporating Arrow as sort of this next-generation in-memory format for processing. Spark also, with 2.3 which is coming out, I think, shortly also has incorporated Arrow as an internal representation for certain types of processing and then Dremio, the product that I worked on most recently, also uses Arrow internally.

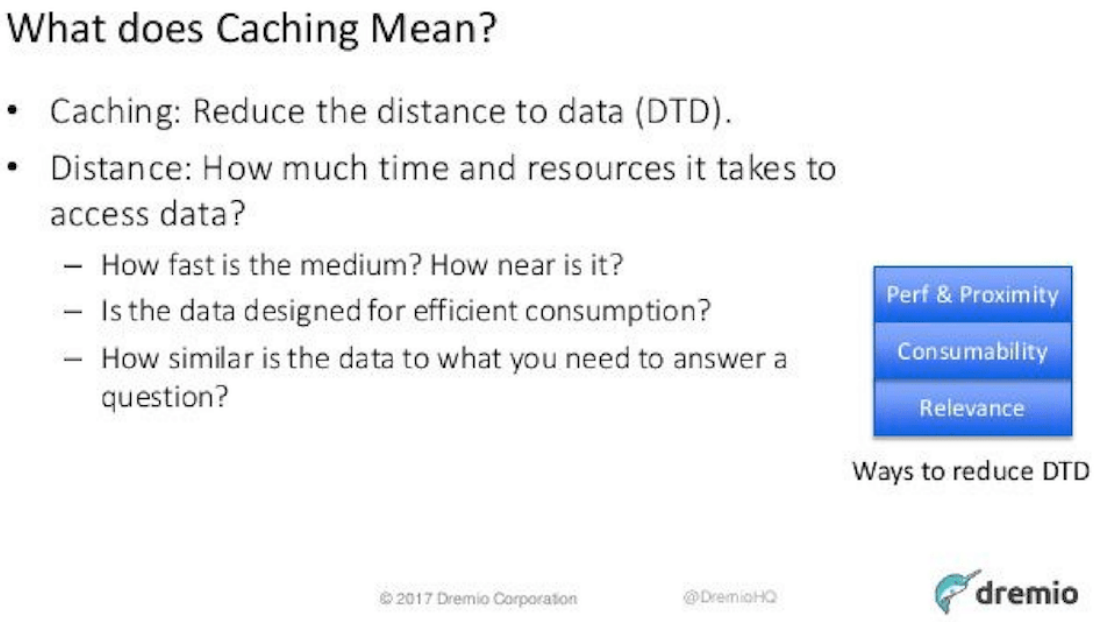
The second piece of ingredient that we're talking about here is Apache Calcite. Calcite has been around for about five or six years as an Apache project. The code has actually been in development for, I think, over 15 years total, so a huge amount of capability here. You can think of it as like a database in a box without storage, but it's actually designed very much like a library so you can use individual pieces to solve specific problems. It includes things like a JDBC driver, a SQL parser, relational algebra as sort of a way to express relational algebra as well as a very, very powerful query optimizer with a lot of common relational transformations.

It also is nice because it understands … You may have caught the talk yesterday if you're on the Data Engineering Track yesterday. Julian Hyde is the creator of the Apache Calcite project and talked probably in more detail about Calcite specifically. I'll talk about it in sort of more general context but understands materialized views and this concept of lattices which is a nice structured way to deal with materialized views, and is used by a huge number of projects, some of the examples are Apex, Drill, Hive, Flink, Kylin, Phoenix, Samza, Storm, Cascading, and of course, I used it inside of Dremio.

The third project I'm talking about is called Apache Parquet. This is probably most likely to be the most well-known of these three projects. It is pretty much, I would say, the de facto standard for on disk representation of columnar data. It was based on, I think, six, seven, eight years ago the journal paper came out. Really, the journal paper didn't talk about a lot in terms of detail, but the one thing it did focus on was an on-disk representation for complex data that was highly efficient and columnar in

structure. Parquet is an open source implementation of that specification, and it supports this high-level concept of different kinds of data-ware compression, three different columns. And also, it's very well optimized for a vectorized columnar readback. It's a de facto standard for on disk.

Talking about caching techniques, I want to sort of start by talking about what does caching mean, and what does it mean to me really. I think that people generally think of it as one thing, but I think it means a lot more than that. And really at the core, what a caching is about, is about reducing the distance to data. It's like how quickly can I get to the data that I need to? Really, that's all about how much time does it take and how many resources does it take to get it to a particular amount of data or type of data.



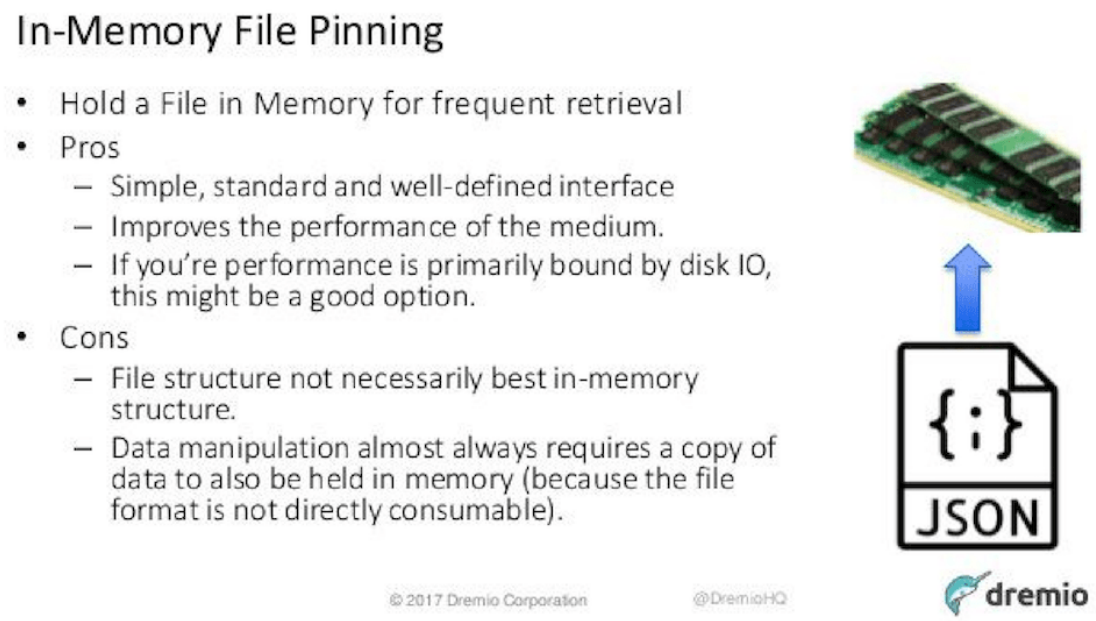
I actually break this into three subcategories. The first is what I can start off here on the right. Here, you can see performance and proximity. That's like how fast is the medium. How near is this data to me? Is it on the same node that I'm on? Is it on in-memory as opposed to on-disk? Is it on the CPU cache instead of in-memory? That is one type of way you think about caching, but there are two others that I think are also important. The second one is this concept of, is data designed for efficient consumption? Maybe that it takes … It's very expensive to process the data, or the data is in a way where I have to skim over a lot of data to get to the specific data that I'm interested in. The consumability of the data is as important as the performance and proximity of the data.

Then the last thing is that … And this is the one that's probably the biggest leap for people, is that you also have to think about how similar is the data to what I need to answer. And I call this relevance. If I need to answer how many people bought a book yesterday in New York City, if the data is all sales of the world, well then that's quite a ways from what I want to answer. But if the data is all the sales in New York, well that's a little bit closer. How relevant is the data to what my question is? When I talk about caching, I mean it's on all three of these different dimensions, is I want to increase the performance and proximity of data. I want to deal with consumability of data and make the data as consumable as possible. But I also want to make sure it's as relevant as possible.

When we talk about different types of caching, I actually break it into six different kinds of caching. And the one that most people think about when they first think about caching is probably the first one. But I'm going to go through each of these and sort of talk about the strengths and weaknesses of each, and as you might guess, in order to solve most real-world problems, it's not like you're going to use one of these. You're probably going to use more than one, and in our case, we use five of the six.

The first one most people know about, in-memory file pinning. HDFS has this capability. It was originally sort of people start to do a lot in a big data ecosystem with Tachyon which also moved and became Alluxio. It's great for a lot of reasons. The first is that it's very easy to understand. I understand that I can take a file instead having it on disk, I have it in memory. It's very simple concept. It's a really well-defined interface like I know how to interact with the file system. I open a file, I read it, I close it. I seek, that's about it. It improves the performance of the medium. "Hey, if I can only pull it from disk at 100 to 150 megs a second if we're using the spinning disk. Now, I'm pulling from memory. I can pull up 10 gigs a second. Okay, that's great. That's going to way improve the thing."

If your performance is bound by your IO throughput, then in-memory file pinning can be very, very efficient and very helpful. That being said, there are some downsides to it too. Well, the first is that the way that you structure a file for a spinning disk, for example, is not necessarily the way you would want to structure it for in-memory. The tradeoffs that you make are also very different. For example, people will frequently use some high-speed compression algorithm to a compressed data when they drop them onto disk because they do math like, "Hey, it's going to take … I can do 500, 600 megs a second single thread of decompression." A disk can only pull a 100 to 150 megs and so I'm actually not bottlenecked at all by that compression. Well, all of a sudden, I dropped that data into memory and all of a sudden the compression actually slows me way down, and so I don't get the theoretical performance of the memory because I've got to go through this bottleneck. You're moving the bottleneck as you do with any kind of performance analysis.



The other one is that if you think about data processing engines and algorithms, typically, they need to use an in-memory representation of data. However, it's not the disk format. Even if you load your data into memory as a pin a file and then you try to do analysis against it, whatever tool you're using will then take the data and make a copy of it in a different representation most likely in memory to do its actual work. You don't actually share a lot of benefit across different application. You share throughput benefits, but you don't share memories management benefits. Everybody still has to make a copy of that data into their own representation. Good for some things, not as good for other things, but the one that most people think about when they think about caching.

The next one I'm going to talk about is columnar disk caching. This is about the idea that, "Hey, you know what? Yes, one way to improve performance is by moving the medium that you're interacting with, but another one is about consumability." If I can figure out ways to improve the medium and the consumability, that's very, very powerful. Most people might land their data in some kind of row-wise format like JSON, or CSV or something more custom, Avro maybe, on disk from a streaming process or logging process or something like that. That representation is not the most efficient for a repeated read. In the case of repeated read, I want to take advantage of something like columnar formats where instead of slicing things by record by record, I slice them column by column.

One of the key reasons that that's very beneficial is that most analytical applications only interact with a few fields in a particular dataset. I might have a data set which has hundreds of thousands of columns but I may only want to interact with ten of those things. If that's a row-wise representation, I have to skip over the rest of the columns every time I want to look at a record. But if it's columnar representation, then I can just read the few columns that I'm interested in, so that's really good for that. They can also be compressed very well because you have data-ware compression and so you can actually reduce IO from both only reading some of the data as well as having very compact data. It also is designed very well for reading back into a columnar representation in-memory and vectorizing that processing.

Now, on its flip side like anything, "Okay, if I'm going to make a copy of the data, that's going to be more expensive than having the original." Now, some people have the ability to land the data directly into a columnar format on disk but it's generally very difficult and the reason is, at its core, is that columnar formats requires substantially more buffering than row-wise formats. A typical, say, a Parquet file, you might actually want to buffer somewhere between 256 megs and 512 megs of data before you spill to disk because you're reorganizing the data at the columnar level.

If you've got lots of streams coming down, then you have to worry about durability. If I'm going to buffer 512 megs of stuff, do I still have to implement a log to have my transactions before I can convert it into this format? It's one of the things where what I see most of the time, I think most people usually so is they land the data in a row-wise format and then later, use some kind of ETL tool to convert it into a columnar format, whether that's Spark or a MapReduce, in the olden days or whatever. The near olden days, I guess, I'd say.

A layer on top of this, in-memory block caching. In-memory block caching is the idea of, "Hey, rather than pinning the whole file in memory, I'm going to pin individual blocks in memory." And the most common way that people are probably very aware of this is the Linux page cache where basically, it's going to say, "Hey, this is a hot block. That's a hot block. I'm going to keep those things in memory rather than having to go back to the disk." Very helpful if you have a data format which is designed for random access and you can get the data that you're interested in, in a small amount of random access chunks. Another example of this is the HBase block cache which is slightly more custom than, say, a generic Linux page cache but it's the same basic concept of … In HBase, it's like, "Hey, I've got a KV store and there's a certain set of keys that are pretty hot, and so, I'm going to keep those in memory."

In generally, very mature especially if you're using Linux page cache. You can get it for free, but you don't have a lot of control or influence over it. It's also very disconnected from the workloads. It's a

random byte range. If those bytes are hot, then great. If those bytes are not hot, then it's awkward. It's actually kind of while it can be beneficial and it's free in a lot of cases, if you look at most analytical workloads which what I'm focused on, you're generally doing large bulk reads. You can actually have problems by blowing out the page cache and that kind of thing.

The next type of caching is going more in the consumability side of things. It's what I'm calling near-CPU data caching. I don't think this is a good name but I don't have a better one. If someone comes up with one, I'd appreciate it. Tweet one at me. It's basically holding the data in a representation that can be processed without restructuring. This is really the core of what I was talking about before with Arrow. It's that, "Hey, I'm going to hold the data in a representation that is ready to be consumed by an application," rather than storing it in a disk format which then has to be interpreted, deserialized or whatever before it can be processed.

If I hold it in a format that's very efficient for processing, then I can potentially have multiple people interact with that data without having to make copies of that data in memory and it also is very, very efficient for doing different works. If I hold data in memory, let's say, near-CPU data representation, then when I want to work on it, I just start working on it. There's no extra work to be done and so that you get 10 gigs out of memory or whatever and then you're immediately working on that as quickly as you can work on it with a CPU. That's cool because then you can have multiple consumers potentially interact with that data.

Now, there are some flip sides to this though. There are some challenges with these as well. Like all technical decisions, there's plus and minus at everything. The first is that this is far larger than compressed formats. A Parquet file, when turned into an in-memory Arrow representation, maybe anywhere from three to ten times larger. It could be even more than that depending on how sparse the data is, how many nodes. One of the reasons is that in-memory representation, something like Arrow, it is designed to support random access and so you don't have to go through the first hundred records to see the 101st record whereas something like Parquet is designed for linear access and so you actually, in order to get the 101st record, you have to skip. You have to interact, read all the first 100 records because you don't know exactly where the 101st record is.

The other challenge with this is that it's very central to an application in terms of how it works with data, and so applications have to be updated to work with this kind of representation. It's very powerful. It can provide a lot of performance, but there's also tradeoffs in terms of size and the level of customization you have to do to work with it.

The last two I'm going to talk about in terms of caching techniques are cube-based relational caching and then arbitrary relational caching. Again, names that I've given these things, you're not going to be able to go out there and find a Wikipedia article on these things. But huge cube-based relational caching is something we'd known about for a long time. Think basically MOLAP. The idea is basically is you pre-aggregate certain pieces of partial aggregations of data, put them in cuboids and then when a user comes in and wants a particular dataset, you could pick what is the right one to use. Very great, provides very low latency for common aggregation patterns, and in many cases, cube storage requirements for common patterns can be vastly smaller than the original data set. Yes, it takes a little bit of resources but not a lot.

Now, the flip side is that for cube-based relational caching is that the interaction speed with data, the ability to retrieve data quickly is very bi-modal. If I get a cube hit, I get answers basically instantaneously or close to it. But if I miss the cube, then I've got to go back to the raw data and who knows how long that's going to take to get back. The other one is that cubes, it's difficult to use a cube to satisfy an arbitrary query pattern. It's great if I'm like, "Okay, I'm doing a Tableau workflow where I'm grouping by A, grouping by B, some in C." If I start wanting to drop into their window functions or arbitrary expression trees and those kinds of things, there's ways to solve some of these things but in general, it breaks down when you have sort of an arbitrary query pattern that you needed for a dashboard or something like that.

That is why you bring in the second concept which is arbitrary relational caching. This is similar idea to the cubing which is basically is I'm going to create an alternative version of data and I'm going to use that in replacement of going to the original source and reconstructing the answer, but I'm going to use it for arbitrary design. Okay, I want to do a window frame, join these two tables and then do an intersection with this other thing. That is an arbitrary query pattern. I want to make that faster because I know that frequently, I'm going to interact with some subset of that data. Arbitrary relational caching, you start to see this triangles, that's what I use to describe these graphs of different operations. You might say, "Hey, I'm going to have this green triangle which is a set of operations I want to solve. But I can solve some portion of it with the purple and the blue triangles."

It's great because it's very simple. It's like, "Oh, this sub-portion of what I'm going to do, I'm going to save that aside and then reuse that rather than having to go do that for every single time." And that also means that you can improve the performance of basically any kind of workload. The flip side is that it's, one, complex to match arbitrary queries. Queries are very, very complicated and there's lots of different ways that they can be patterned. Being able to recognize that this is roughly the same as … Or this can be replaced with this is actually a fairly complicated problem. And the other thing is that arbitrary caching can be large. An arbitrary cache could be a copy of the whole dataset, and if I have

a copy of the whole dataset, well of course, I'm going to be paying twice the storage costs. But it does allow you to fit your caching to a particular business use case no matter what the pattern is there.

I've got a little feedback on this. We'll see if you guys think it makes sense. Give me a red card if it doesn't make sense. But this is how I think about how these things all fit together. You've got the two axes. The performance and proximity is on the Y axis, and then you've got the distance between representation in question on the X-axis. In the X-axis world, you have the consumability dimension I was talking about before and then the relevance dimension. The perfect situation, the best possible version of a cache is something that is highly performance very proximate to your CPU, is very, very consumable and is highly relevant to your question. Now, of course, that is a very custom place in the graph. We know that the cost of mediums go up as they become more performance and so they're going to be very expensive to put stuff there. And we also know that the more you move across on the right on the X-axis, the more specific your answer is.

If I have a raw data set which has every fact in the world, then that dataset can answer every question but it's a long ways from answering a particular question. If I have an answer to one question, then I know I'm going to be able to answer that question very, very quickly but it's only going to satisfy a small amount of things. It's very hard to get to that top right because you have to worry about both the applicability of what you're doing and the scarcity of the resources that's required to be able to get at it.

This is how I lay these things out. In-memory file pinning, in-memory block caching, very good at being performant and proximate, but really have no influence on the consumability of things or the relevance of things. You can go out into the right with near-CPU data caching which gives you improved consumability and also performance and proximity by setting that in memory, but you also know you're going to use way more and more space than you might use if you're doing something like in-file pinning. Columnar disk caching on the other hand, generally speaking, it's a disk cache so it's not going to be as fast as in-memory but it's going to substantially improve consumability and also not used that many resources.

Now, what gets really exciting is we need to look at the two kinds of relational caching. Relational caching starts to move you out on this relevance curve which really allows you to make magnitudes of difference in terms of performance whereas many of these other techniques, you get 2x, 5x depending on what your particular use case is. Maybe you can find a use case that's really awesome but you're not usually going to get magnitudes of these things. You're just going to sort of be, "Hey, I can scale out my system or whatever to improve latency." But when you use cube-relational cache

and arbitrary relational caching, you have the ability to really do a good job of solving the relevance problem and therefore, substantially reducing that distance to data.

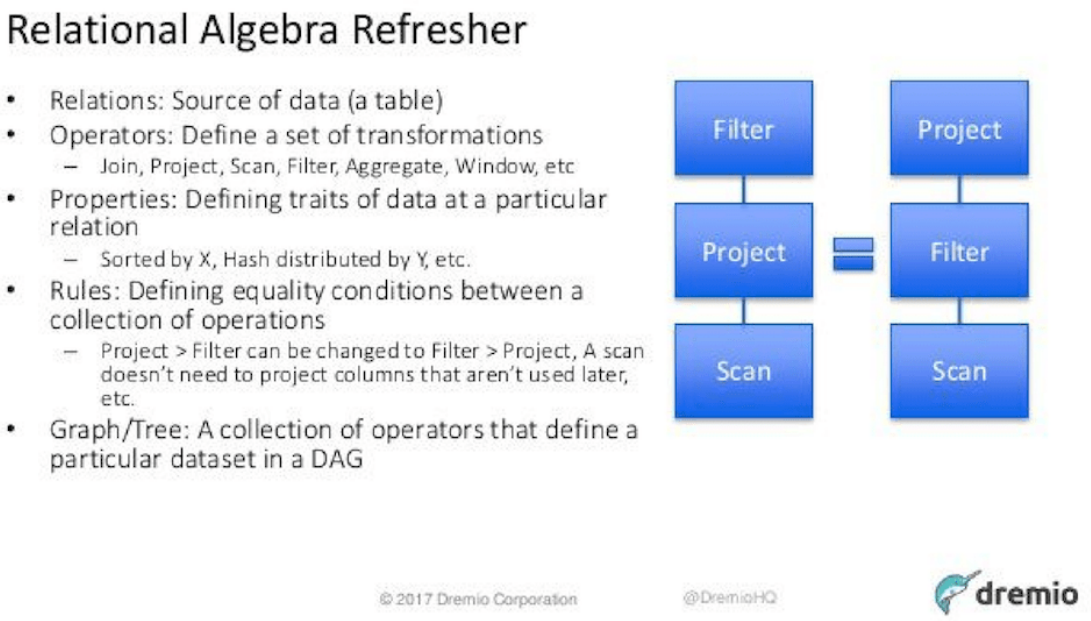
What we built at Dremio, we focused on all these different things and started to evaluate what are the things that we think make the biggest impact to people's workloads. We actually decided to take some combination of five of the six of these. We actually don't do in-memory file pinning but we actually put together the other five to try to solve the problems that people have.

Let's talk about those two last categories which is relational caching and how that works specifically. I'm going to start out with a very … This is not a lot of content on this but a very quick relational algebra refresher. Relational algebra is a way to express how you work with data. That's very generic, I know. But it has these key concepts, relations, so a source of data. I'm like going to get a data from a particular table whether that's on [inaudible 00:22:46] CSV file on disk or a relational database table or a collection MongoDB, whatever the case may be.

Then, you have operators, and operators are basically transformations. How do I go from one state of data to another state of data? Then you have properties or were sometimes called traits. And a property of data is some nature to the stream of data as it's moving out of that operation. If I sort

some data, then afterwards, it has a trait or a property which is sortedness. It may have a property of distribution of the data inside of a distributed cluster.

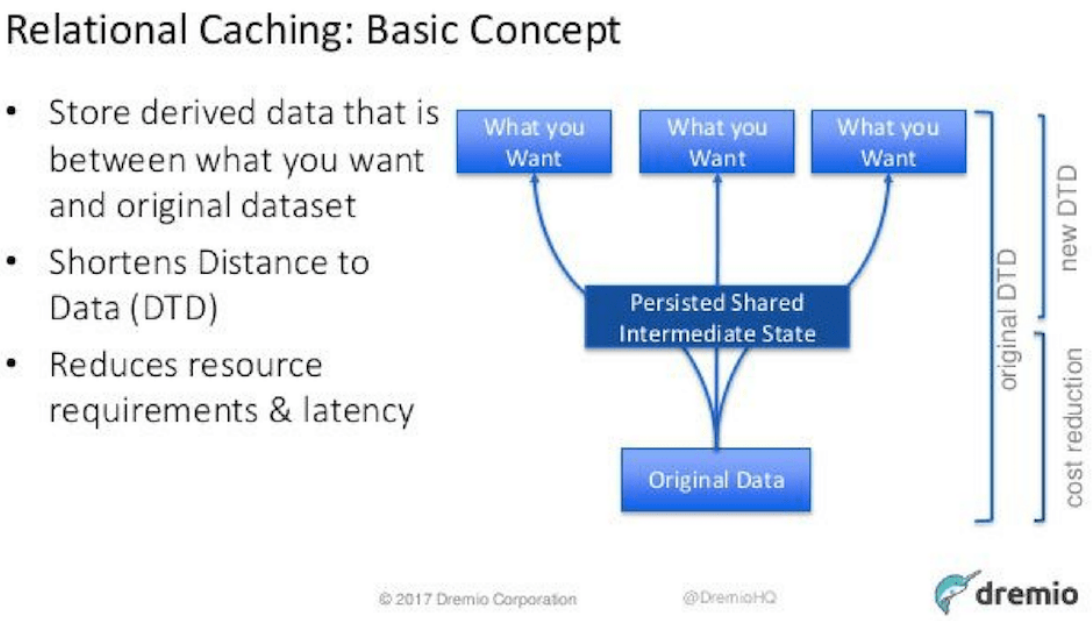
You then have rules. Rules define basically how you can manipulate a set of operators from one representation to another representation and those things are equivalent. The whole concept of working with relational algebra is that I can take a tree that looks like one thing and do a bunch of changes to it and I know it has the same meaning. I'm going to get the same data out of the first tree and the second tree but the second tree is somehow substantially better for performance in terms of how I'm doing the work. Then lastly, basically, I talked about graphs or trees which is a collection of these operators that are collected together with relations at the leaf nodes and basically, moving this data into a representation that's relevant to me.



When we talk about relational algebra, one of the common things that we want to do is we want to do some kind of transformation and say that these two things are equal and a very simple version of this might be what's on the right-hand side of the slide which is that originally, I have something which is a scanning of data followed by a projection. I only want three of the columns of data followed by a filter. Maybe as a projection, maybe I want three of the columns. I want A, B, C and then I also want to add D and E together and make F.

Then, afterwards, I want to filter by A that is greater than 10. That might be a very simple relational algebra expression that's saying, "Hey, this is how I want to get to a particular data set." You can have a rule which says, "Hey, you know what? Actually, it makes sense. Before I do the addition of D plus E making F, I can actually filter the data on A and that means I can do less work of the D plus E equals F. I have to do less of those because I filtered out some portion of the data. That's a very simple relational algebra transformation. I'm going to change the project and the filter so that the filter comes before the projection and that's going to improve my performance.

That's relational algebra refresher, very quick. Relational caching is basically saying, "Okay, now that I see that there are some commonality to different things, I want to hold some intermediate state." You have some original data and you have different things that you want. I have what you want A, what you want B and what you want C. Normally, if I wanted to answer each of those things, I would say, "Hey, I'm going to go and have to go from the original data and I'm going to have to derive what I want." That's a certain amount of distance to data. But if I realized that those three what you want have some intermediate shared state which is closer to the answer but is not the answer, if I can persist that state and use that instead, then all of the sudden, I've substantially reduced the distance to the data for the individual questions. That can reduce my latency. It can also reduce my resource requirements.



That is the basic concept of relational caching and saying, “Hey, I’m going to get something that’s closer but not the answer and use that for multiple different situations.” Now, as you may be thinking already, I already do this. Most people do this already and what that means is that you know what? I may have some raw data that was in JSON. I converted it into Parquet file. I’ve got a dashboard that I need to have supported and so I’m going to then convert that into a summarized version of the dataset. I might sessionize the data, cleanse it. I might partition it by time or region. I may summarize it for a particular purpose like a dashboard. Then, I have the users or the developers, pick which one to use. They may say, “Oh, okay, I know that I’m doing a longitudinal analysis so I’m going to use dataset X, version X of the dataset and, oh, now, I’m doing a Tableau dashboard so I’m going to use a version Y of the same dataset.” This works. As long as you don’t have too many choices, it can work pretty well.

What happens though is that as you add more and more choices, you get into an awkward situation and this is where relational caching really comes into play is that normally, an end user comes into … Your analysts or your end users, your developers, data developers come in and start trying to figure out which datasets to work with, you basically have to train them on what’s the right one to use. That can be very challenging for users to learn implicitly, “Oh, this is the right one or that’s the right one,” all the time especially because if they’re working through some kind of analysis problem, they may start with one dataset. And as they’re working and building up queries on top of queries on top of queries, it may make sense for them to actually switch to another dataset in the middle of that work but they’ve already built up this whole stack of SQL queries to get to what they want to, and the odds that they’re going to take the inner part of that query and change it out with something else is very, very low.

People, all the time, use what I would call a copy and pick strategy, which is I’m going to copy the data or drive a piece of data and then I’m going to have the users pick the right one based on their use case, which does solve a bunch of these problems but it becomes a bigger problem when you have hundreds or thousands of these choices that an analyst has to make, and the analyst is not necessarily incented to even make the right choice. They don’t necessarily want to spend the time to think about this.

Relational caching is this concept of saying, “Hey, you know what? Let’s make the cache pick the right version of the data. Usually, you can always interact with the raw data or some virtual version of the data but the system can then make the choice of which physical representation of the data is most satisfactory for solving that as quickly as possible. It’s moving the responsibility of the pick and copy in the cache. The second part is it’s also moving the maintenance of those copies to the cache from having some data engineer or data admin who is responsible for maintaining all these representations. It’s something you already do today, it’s just saying, “Hey, let’s make the system do

this rather than making end users do it because the end users is going to make mistake or not necessarily have the time to think about these things."

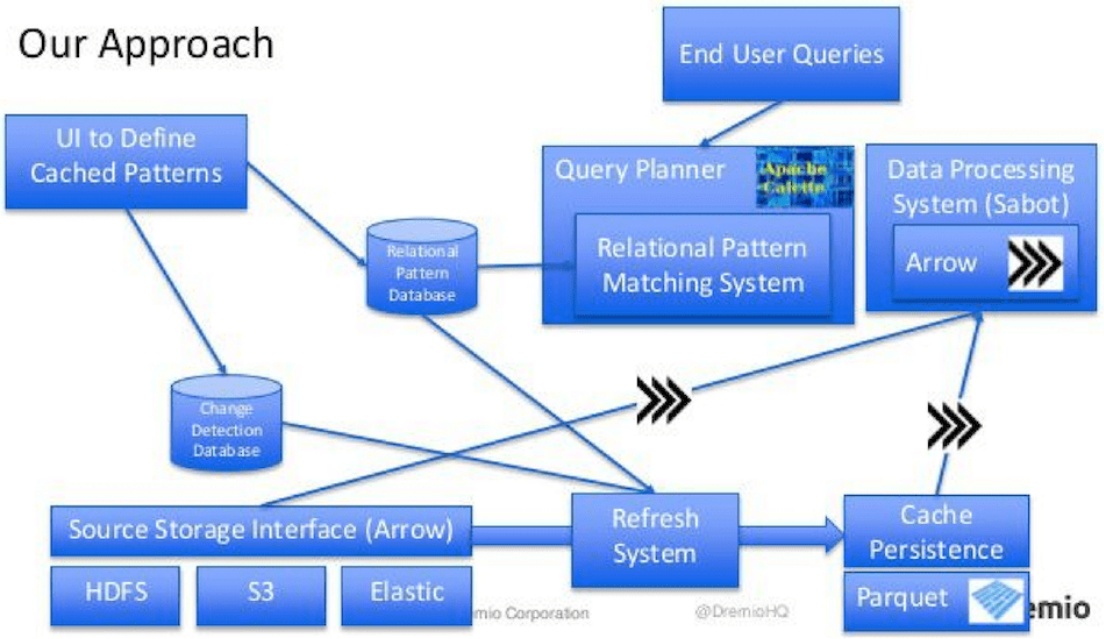
Building a relational cache as we did, we started with, "Okay, what are the key components?" The first is basically how do you express transformations and states? There's multiple ways you can do it. You can do it code base. We felt that SQL made the most sense. It fits to at least a lot of the people's understanding of how to express different transformations. Let's use SQL. Then, okay, once we do that, how do you hold and manage these relational algebra representations? There, that's when we took Calcite and we said, "We're going to use Calcite to solve this because it's a very good toolkit for that."

Then, you said, "Okay, well, I'm going to cache these different versions of the data. I want to persist that." How do I persist that? Well, that's when we use Parquet. We said, "Okay, let's use Parquet to persist it," and then there's a need after you persist these datasets to separate what the cache version of the data is to the actual answer. If you go back to this picture here, we still have the second part of the top. You see that up there? Yeah, so we still have this part of the top that we have to do. We've cut out the bottom part here. We don't have to do that anymore but we still have to do this top part and so, we need something to do the top part. Then, we say, "Okay. Well, the way we can do that is we can use Arrows in-memory representation processing," and then we built an open source engine on top of that called Sabot which is designed to process that quickly and bridge the gap between what the data representation is and what the actual question is that the person is trying to answer.

There's all a lot of other code that get put in there, you can go look at it in our GitHub, but this is how it came out. Basically, it's just you're going to have a UI to define your different cached patterns and that's going to interact with two different things. It changed the detection database and the relational pattern database. Relational patterns are very useful because then you might have an end user query that come sin. You go in and you're going to use the query planner, in this case, Calcite and we have a relational pattern matching system that's built inside of Calcite that is responsible for figuring out what are the right relational patterns that can help to solve this particular question. That is then handed into a data processing system which is the Sabot and that is interacting with Arrow representations to work with data.

Down below, you have different kinds of data that you're interacting with, whether that's data in HDFS, S3, elastic, relational or whatever. Then, you have a source storage interface which is going to be producing Arrow data and bringing that into the processing system. But then you also have this other thing which is the refresh system. The refresh system is responsible for reading data out of these

individual systems based on different kinds of change detection and moving that into the persistence layer which a Parquet representation of data.



When a particular user question comes in, we may actually interact with the cache data entirely and we realize that we can satisfy the entire answer by taking cache data and then further transforming it or maybe we realize that you can't satisfy with cache data and you have to go back to the source or more likely, you actually have some combination of the two. Maybe that I have some portion of the data cache, maybe my fact table has been aggregated down to a much smaller size and I can use that, but we'll still go to a dimension table inside of my relational database rather than caching that data. Basically, being able to combine cache in some situations and not cache in another situations.

All right, I'm going to go to … I only got, I don't know, 15 slides but we'll go quick. Definition and matching. Okay, this is what relational caching is trying to do, how do you do it? This is coming back to Calcite. Calcite is this great toolkit which has all these different ways of expressing how these things work together and it comes with all these pre-built tools like here's all the operators that I might want to have. Here's a bunch of rules. Here's a bunch of properties. Here's I could transform between two things. It also has a really cool materialized view facility. For those of you who have come from relational database background, well, relational caching is basically materialized views revisited so that they're outside the database.

Now, they're also enhanced in various other ways but that's the way you can think about it. Hey, look, Calcite becomes a great foundational component for building a relational caching system. How we think about it is we have to come up with entities. You have to come up with ways of thinking about the individual objects that's useful. We came up with this concept of reflection. Reflection is basically a persisted version of the data that may satisfy some alternative portion of the original dataset.

We persist those in a Parquet format and then we have two main sub-concepts of reflections. We have what called raw reflections which is basically a persisted version of the dataset that includes maybe some of the columns or all of the columns but is partitioned and sorted in a particular way. My dataset might be initially in a JSON representation. I'm going to say, I'm going to create a raw reflection of that dataset where I only include half of the fields and I'm going to partition it by date and I'm going to sort it by transaction amount.

That's raw reflections. Aggregate reflections is the second concept which is more of the cubed view of the world which is to find dimensions and measures on a dataset and say, "I want to create various levels of roll-ups or partial aggregations against those dimensions and measures. Hence, you can still in that case, control both partitioning and sortedness.

These two tools are actually very powerful because they can always be layered on top of virtual datasets. Yes, I can create a raw reflection against a dataset which is just the source dataset but I can also create a raw reflection against some arbitrary set of transformations I've created and so, I want to do a window frame. I'm going to do an intersection and then I want that to be a virtual dataset and then, I can apply a raw reflection on top of that and make that be the persisted dataset and I can still make those choices around partitioning and sortedness only including some of the columns so that I can decide how … You can think about these raw reflection like a covering index of either a virtual or a physical dataset.

Aggregate reflections are more like cubes. These are all built to allow us to do what is basically decisions around using alternative versions of the data. As an example here, we may have a query which is … I'm using abbreviations over here to make this compact but basically, we're scanning some data. We're doing a projection of only a couple of columns of the data. We're then doing an aggregation and then we are going to be filtering some data.

This is what the users comes in. He says, “I’m going to interact with raw data. I’m going to do these operations.” Well, we define a reflection in the middle of this world which says, “Hey, I’m going to say that X is equal to Y here.” That’s target is equal to materialization. If a user has this part of the query in their query, it may not be the queries exactly that but part of their query can be turned into this, then instead of using that target, I can actually use a materialization that I created which is that work already done.

What happens here is that I may have originally wanting to have to do an aggregation against a large fact table but it turns out I have another aggregation which is covering the aggregations that I’m asking for. Instead of using the aggregating as the raw table, I’m going to aggregate against this materialization in this reflection instead and therefore, I’m going to do another plan. Now, the plan doesn’t look less expensive when you look at it just as blocks because it’s three blocks versus four, who cares, but the reality is in this case is that the scan in this case can be several magnitudes less amount of records. The overall amount of work here is far less.

This is an aggregation roll-up matching cases and you’re going to say, “Hey, I see this pattern in the query. I’m going to replace it with something else.” There are other things like this and there are lots and lots of these different kinds of patterns. Another example is a join/aggregation matching pattern. You transpose the two and so, it may be that my raw query here, my original query is an aggregation on top of the join of two tables. Well, it may be that I have a materialization which is an aggregation on top of one of those two tables, not an aggregation on top of both of those two tables.

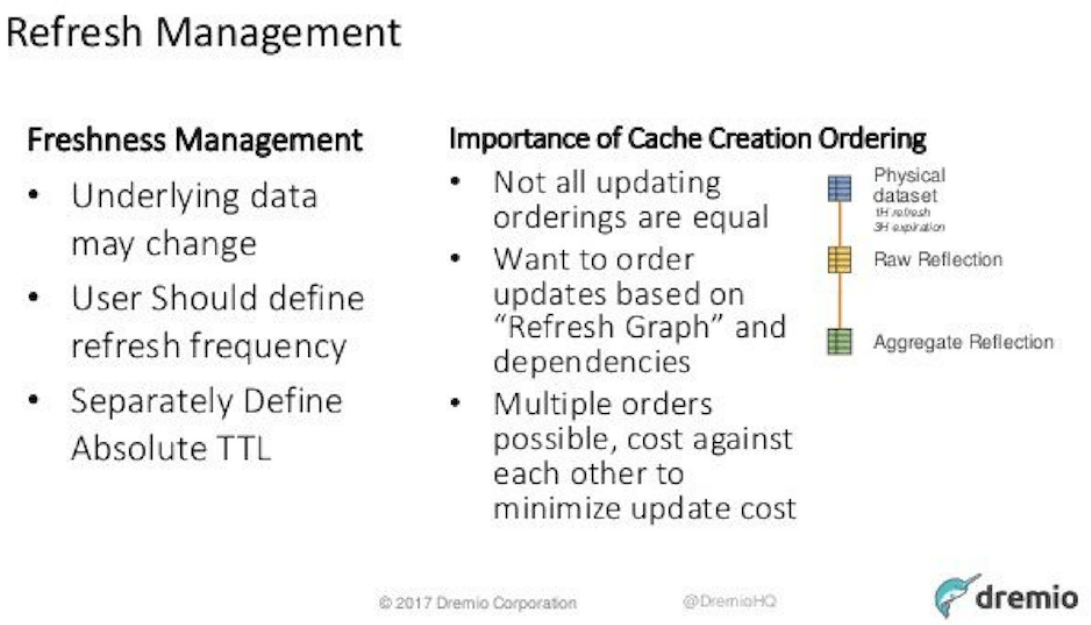
Well, there’s a bunch of rules that you can use. Just swap these things around and actually come up with a tree that uses that pre-aggregated dataset on one side and do some other work on the other side to come up with the right answer. All of the sudden again, you substantially reduce the amount of work. There are lots of these different transformations. One more example is that there’s actually not one transformation. You’re going to actually come up with many. You may come up with multiple replacements simultaneously. It’s not only about coming up with what matches. It’s coming up with the best match.

Here’s an example where I might have two raw materializations or two raw reflections, one which is partitioned by A and one of which is partitioned by B. A user comes in, wants to do a scan of data and then filter by A. Well, as you might guess, it’s much better to use the materialization where I’m partitioned by A because then I can just quickly prune all the partitions and never actually have to apply the filter at run time as opposed to use the dataset which is partitioned by B.

All of the sudden, I get a new plan which is the scan of just the raw data, this partitioned by B part by A but only the valid A's that are relevant to the query. Lots and lots of different patterns like this and we can talk more about those offline, come and chat with me about it. But there's lots of different ways of approaching these different matching problems to solve, "Hey, how can I solve this question with this other answer."

Lastly, just a quick couple of slides on dealing with updates. As a cache, you have to think about updates too and maybe in a perfect world, there would be no change to your dataset from the [census 00:37:23] data from 2010 and so I don't have to worry about any changes, that would be great. But most of us don't work with that kind of data. We work with data that's changing all the time. When we thought about that, we solve it in two different ways.

The first is we thought about refresh management and so, two sub-components of this is freshness management. When you're interacting with the data as an end user, you want to be able to know that the data is no older than something. You need to have a guarantee that the data is no older than something but you also want to probably achieve something that's more fresh than that. Worst case scenario, it's no older than three hours but I like it to be updated every 10 minutes. That's how we broke it down is you can set a refresh time but just how often you want the data updated but if your data system is really loaded or the data hasn't showed up yet, then you can set a tolerance for how old the data can be before we actually start serving directly from the source rather than from the cache. You can say, "Okay, I want to set an absolute TTL expiration date to the data as well and decide between those two things.



The other thing to think about is that when you are updating these different representations, there is a good way and a bad way to update. Well, there's lots of ways to update them but most of them are not as good as the best way. The idea is that you probably are going to create multiple relational caches of the same data at once. And what you want to do is you want to calculate what's the optimal order of updating all of these things so that I can do a minimum amount of work because if I have, for example, a raw reflection and then I have an aggregate reflection, the raw reflection which will be columnar and then the aggregate which is going to also be a columnar, it's a lot better to do the raw reflection first because now, I have an optimal representation that I can do the aggregate reflection on top of rather than say, going back to the JSON source twice.

The other thing that we focused on was building basically the dependency graph of the different relations to figure out what's the optimal way that we can refresh all these things to minimize the amount of overall touch time. This is one of the big benefits of having a system do this rather than the end user. End users try to do this but they can't do it nearly as well … Not end users, data admins or data engineers try to do this because they say, "Oh, I know this is better to come before that," but it's still very difficult to figure out all those relationships and do that well. Instead having a system do this actually makes things a lot easier.

Then, lastly, you think about updates themselves, mutations. How do you deal with the mutations? We basically came up with four main ways that we deal with mutations. The first is what we call the full update which is the most obvious. It's like I'm just going to copy all the data again. I'm not going to worry about the mutations and the reason that this is important is that some data is mutating so fast they're trying to track the individual changes, it's not going to make sense to try to track those back.

But in many other cases, mutation patterns can be more predictable and so the two that are usually the most common for large datasets are basically a pendulating pattern and that might be for files. Hey, as new files and new directories show up, I'm going to take advantage of that, or as if you've got a rowstore like relational database or something like that, I may have a column which is a constantly increasing column and so I'm going to use that column to identify new records that have been inserted. Those are two ways to deal with append-only situations and so then, you can say, "Hey, I want to get a snapshot of everything that's changed since the last time I interacted with the data and just make incremental updates to individual reflections based on that.

Then, the last is actually a partitioned refresh which we found actually the most useful and partitioned refresh is the concept that I'm going to break my data into individual partitions and sometimes, some of the partitions are going to mutate but it's very rare that all the partitions mutate. Typically, only the most recent partition is mutating if you use partitions of time which is what most people do. This allows you to constantly update one portion of the data but maintain the rest of the data. You can have a moving dataset of the last 30 days and you can refresh it every 10 minutes but every refresh is only going to interact with the partition from today.

Closing words, using these techniques, we've seen … I'm not using benchmarks here but we have seen huge amounts of impact by using techniques as you might guess. If you get the data as close as possible to the question, then you can be able to improve the performance substantially and at the same time, you can vastly reduce the amount of resources. And in many cases, this is the interesting part that's not necessarily intuitive is that we, in many cases, actually reduce the total amount of disk consumptions required for use cases and the reason is because are already doing these techniques but they can't be as good at it as a computer.

While we may actually create more reflections, we can realize that certain reflections are more relying on each other and avoid having to create duplicates of things that are basically the same. Whereas in enormous organization, two different users may have different use case which is substantially overlapping and you're going to make two different copies of that data for that purpose.

Find out more and get involved, whatever you like to do, I'm doing office hours in the East Room right after this, the East Room Lounge. It said East Room. East Ramp Lounge, not East Room, sorry. There's a Dremio table behind you, not behind me, where my coworker is giving some overviews of what we do at Dremio and the product there so you can see what that looks like. We're actually doing an Apache Arrow meetup tomorrow in Midtown at Enigma Tech. Thank you, Enigma Tech. If you want to come and join us, we'll have Wes McKinney who's the creator of Pandas there, talking about Arrow as well as I'm doing a talk, this deep technical talk on Arrow.

Join the community, so there's Dremio community. We've got a GitHub site. We can go download a product and try it out. Come and talk to us about what's going on with that. I'm very engaged in a bunch of these open source projects, Arrow, Calcite and Parquet so go and join those development channels, go and join those mailing lists and check those things out. That's all I got. Thank you so much for your time, guys.