A Graph-Based Firebase

In A Database in the Browser, I wrote that the schleps we face as UI engineers are actually database problems in disguise [1]. This begged the question: would a database-looking solution solve them?

A few months ago, my co-founder Joe and I decided to build one and find out. This became Instant. I'd describe it as a graph-based successor to Firebase.

You have relational queries and basic auth. Optimistic updates come out of the box, and everything is reactive. It is an MVP you can play with today.

Working on Instant has felt like an evolutionary process. We picked constraints and followed the path that unfolded. This led us to places we would never have predicted. For example, we started with SQL but ended up with a triple store and a query language that transpiles to Datalog.

What were these constraints? Why triple store? What query language? In this essay, I'll walk you through the design journey — from problems to solve, to choices made, to what's next.

I hope by the end, you're as excited as I am about what this could mean for building apps and the people who use them.

Delightful Apps

Our journey starts by looking at what exists today. Think about the most *delightful* apps you've tried. What comes to mind? To me, it's apps like Figma, Linear, and Notion. And if you asked why, I'd say three reasons: Optimistic Updates, Multiplayer, and Offline-Mode.

Optimistic Updates

Once you're in the flow of Figma or Notion, you rarely see a loading screen. This is because every change you make is applied instantly. It's painful to do this well. You need a method for applying changes on the client and server. You need a queue to maintain order. You need undo. And the edge cases get daunting: if you have multiple changes waiting and the first one fails, what should happen? You need some way to cancel the dependents [2].

Challenging to build but transformative once done. Interaction time changes how you use an application. Get fast enough, and your fingertips become your primary constraint. I think this is the key to unlocking flow. [3]

Multiplayer

Speed itself is delightful, but it's taken further with multiplayer. Every feature in Linear is collaborative by default. Assigned a task? All active sessions see your change. [4]

There's a pattern to multiplayer too. Developers think it's a nice-to-have. But then some company builds it, and we're stunned by the result. Figma did this for Sketch, and Notion did this for Evernote.

But most apps aren't multiplayer. This isn't because we've hit a sweet spot of text editors, task managers, and design tools. Multiplayer is just too hard to build. [5]

Offline-Mode

Finally, delightful apps work offline. Some not *completely* offline, but they all handle spotty connections.

And offline-mode has the same pattern as multiplayer. It feels like a nice-to-have, but build it and you leap past your competitors. Why? Two reasons:

First, though internet connectivity is abundant, there's a tail end. The subway, the airplane, the spotty cafe. Seems minor, but eliminating the tail-end can be transformative. When we know that an app will work no *matter what,* we use it differently. [6]

Second, your app becomes *even faster*. Offline-mode amortizes read latency. For example, the first time you load Linear, it may take time to fetch everything. But then, subsequent loads feel instant; you'll just see offline data first. [7]

Applications from The Future

Combine these features, and you get an application available everywhere, as fast as your fingertips, and multiplayer by default.

Compared to the average web app, this is a difference in kind. Linear is so fast that you fall into flow states closing tasks. No one would say this about Jira. Notion's offline-mode lets you store every note there. People don't do this in Dropbox Paper. In Figma, two designers can collaborate on the same file. This was unheard of in the days of Sketch.

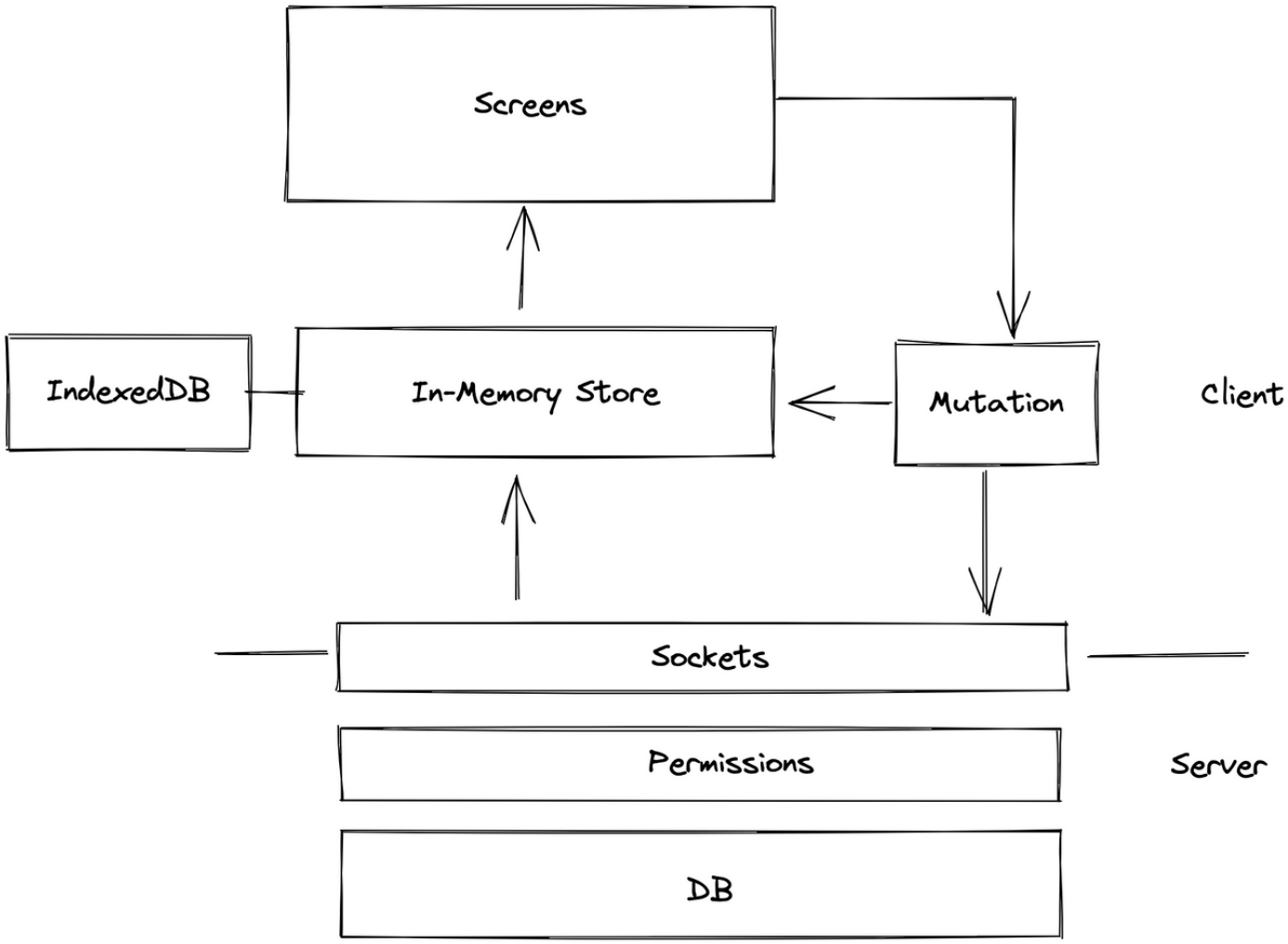
These applications let you work in new ways. They become tools that you can master. And I think this is how most apps will be in the future. We prefer the experience, and the Notions of the world teach us to expect it.

As an industry, we'll need to find new abstractions that make building apps like this easy. I think it's worth the effort to find them now.

Bespoke Solutions

So let's try to discover this abstraction. What works today? Linear and Notion exist; how do they do it?

Thankfully there's lots [8] of [9] interesting [10] work [11] that explains their architecture. Here's a simplified view:



Let's go bottom-up:

A. DB

On the backend we start with a database. Users want a live view of some subset of data. We can keep live views by either polling the database or leveraging a write-ahead log. [12]

B. Permissions

The DB gives us a set of results, but we can't just send this data up to users. We need to filter for what they are allowed to see.

So we build a permission layer. This starts simple. But as an app gets complex, permissions resemble their own language. Facebook had the best design I've seen. Here's how it looked:

```
function IDenyIfArchived(_user, task) {
  if (task.isArchived) {
    return deny();
  }
  return allow();
}
// ...
{
  "task": {
    read: [
      IAllowIfTeamUser,
    ],
    write: [
      IDenyIfArchived,
      IAllowIfTeamUser,
    ],
  }
}
```

Developers write a set of IAllow or IDeny rules per model. Since all reads and writes go through this layer, engineers can be sure that their queries are safe. [13]

C. Sockets

Now we reach the websocket layer. Clients subscribe to different topics. For Notion, it could be "documents and comments." Or for Linear it could be "team, task, and users."

Backend developers hand-craft live queries to satisfy these topics. There's a balancing act to play here. The more complicated the query, the harder it is to keep a live view. [14] So we need to simplify queries as much as possible. Most often, this means we skip pagination and overfetch. [15]

D. In-Memory Store

Now we move to the frontend. Sockets funnel all this data into an in-memory store:

```
const Store = {
  teams: {
    teamIdA: {...}
  },
  users: {
    userIdA: {...}
  },
  tasks: {
    taskIdA: {..., teamId: "teamIdA", ownerId: ["userIdA"]
  }
}
```

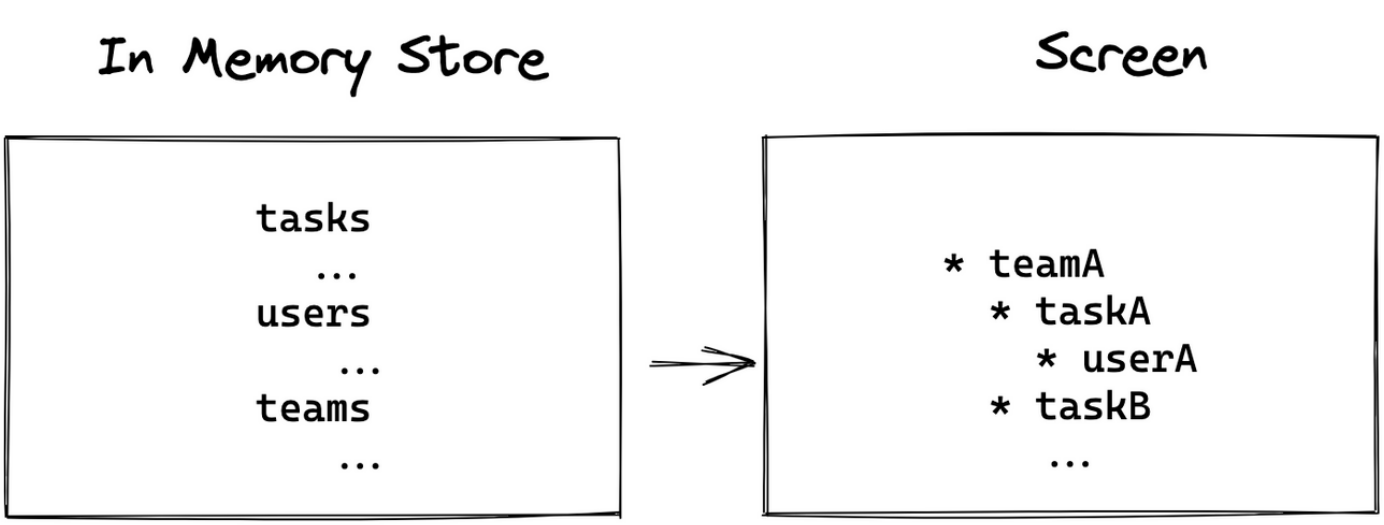
We do this so all screens have consistent information. For example, if a user changes their profile picture, we should see updates everywhere. The best way to do that is to keep data normalized and in one place.

E. IndexedDB

But we need our app to work offline too. So we back our store with durable storage. For web this is IndexedDB. When our app loads, we hydrate the store with what was saved before. This is what enables offline-mode and amortizes read latency.

F. Screens

Okay, time to paint screens. Right now we have a store with normalized data. But normalized data isn't directly useful for rendering. What a screen wants is a graph. Say we show a "team tasks" page in Linear; we'd want team info, all the tasks for the team, and the owner for the task:



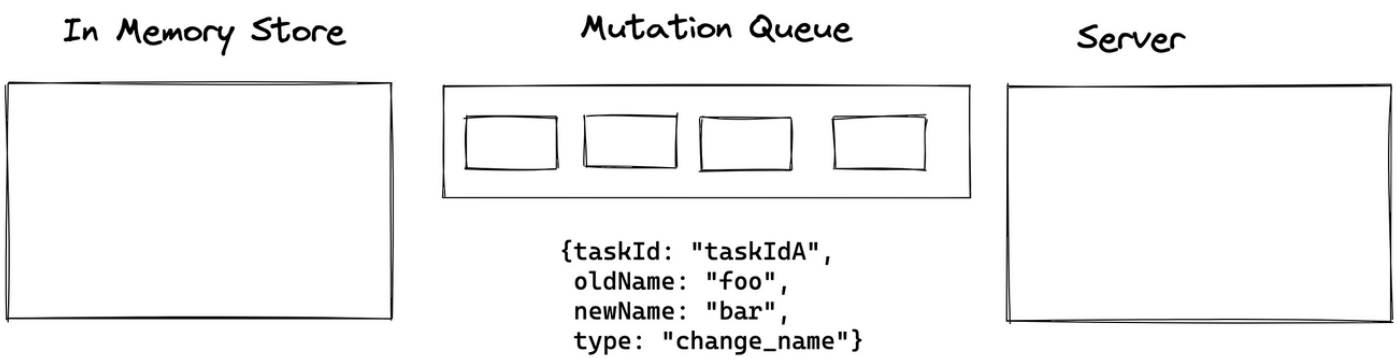
We can build this with a javascript function:

```
function dataForTaskPage(store, teamId) {
  return {
    ...store.teams[teamId],
    tasks: store.tasksForTeam(teamId).map((task) => {
      return { ...task, owner: store.users[task.ownerId] };
    }),
  };
}
```

If this causes too many re-renders, we can memoize it or use some kind of dirty-checking. With that, we have a page a user can interact with.

G. Mutations

Then users make changes. We want those changes to feel instant, so we support optimistic updates. This is how it usually looks:



Whatever mutation we make, our local store and server need to understand them. This way we can apply changes immediately.

To do this well, we need to support undo. We need to maintain order, and we need to be able to cancel dependent mutations. Hard stuff, but Linear, Figma, and Notion all go through the schlep.

Once this is done, we've got an application from the future on our hands.

What Exists

Oof. Lots of custom work. Could these apps have used an existing tool instead?

Firebase

Firebase comes closest. It has optimistic updates out of the box. It supports offline mode and is reactive by default. But, I think Firebase has two dealbreakers: relations and permissions.

Relations

The biggest dealbreaker is Firebase's query strength. You're limited to document lookups. When Firebase was built, this was a great tradeoff to make. It's simpler to support optimistic updates and offline mode for document stores. But for sophisticated apps, you *need* relations.

Figma, Notion, and Linear all have relations. Notion has a recursive model where blocks reference other blocks. Linear has users, tasks, and teams. Figma has documents, objects and properties.

If you need relations, document stores explode in complexity. You end up having to implement your own joins with hand-tuned caches. Another schlep.

Permissions

The second dealbreaker is Firebase's permission system. [16] Firebase Realtime has a language that looks like a long boolean expression:

```
auth != null && (!data.exists() || data.child("users").hasChild(auth.id));
```

This gets unmaintainable fast [17]. It improved in Firestore — there's now a function-like abstraction:

```
function isAuthorOrAdmin(userId, article) {
  let isAuthor = article.author == userId;
  let isAdmin = exists(/databases/$(database)/documents/admins/$(userId));
  return isAuthor || isAdmin;
}
```

But again, this wasn't built for complex use cases. There's no way to write an early return statement for example. If we're aiming for Linear, Figma, or Notion, we need a system that can scale to complex rules.

Supabase, Hasura

So Firebase won't work. What about Supabase or Hasura?

They solve Firebase's greatest dealbreaker: relations. Both Supabase and Hasura support relations.

But they do this at the expense of a local abstraction. Neither support offline-mode or optimistic updates. Multiplayer is still crude. You write basic subscriptions and manage the client yourself.

Supabase and Hasura also don't have a powerful permission system. They use Postgres's Row-Level Security. Permissions are written as policies. But this won't work for sophisticated apps. You'll need to write so many policies, that it'll be impossible to reason about. It'll get slow too — the planner will struggle with them.

The Missing Column

So Firebase has a great local abstraction, but no support for relations. Supabase and Hasura support relations, but have a poor local abstraction. Put this in a table and you have an interesting column to think about:

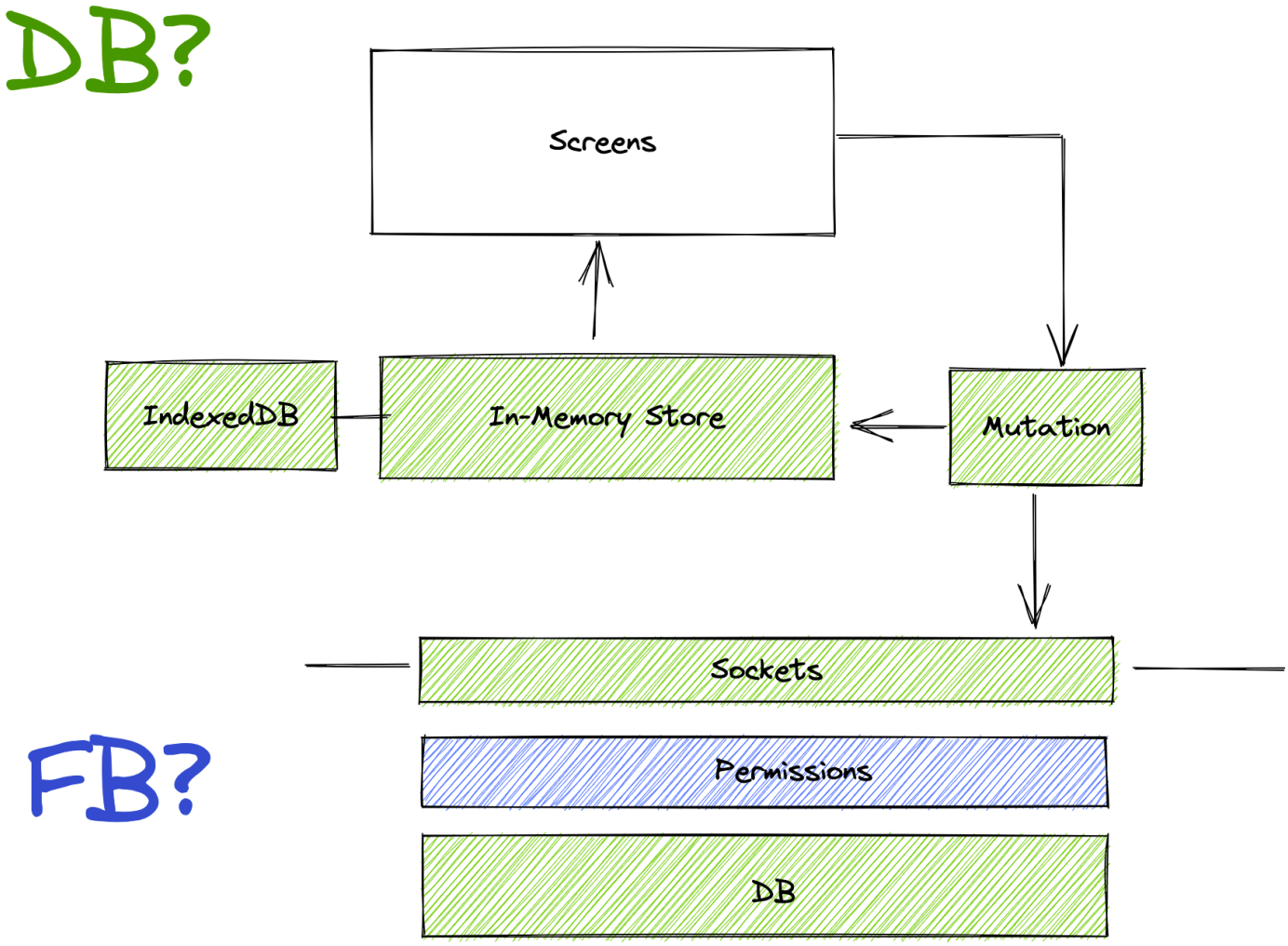
	Firebase	Supabase	Instant? 🤔
Relations	❌	✅	✅
Local Abstraction	✅	❌	✅

What if a tool could support relations and a local abstraction? You could write any query that a Figma, Linear, or Notion would need. And you could handle all of the hard work they do locally: optimistic updates, multiplayer, and offline-mode.

Add support for complex permissions, and you have a tool to build applications from the future!

Inspiration

A daunting column to satisfy. But again, if we look at how Figma, Linear, and Notion work, we find clues. Squint, and their architecture looks like a database!



Again, screens need consistent data. Previously, we wrote functions and got data from the store. Remember `dataForTasksPage` ?

```
function dataForTaskPage(store, teamId) {
  return {
    ...store.teams[teamId],
    tasks: store.tasksForTeam(teamId).map((task) => {
      return { ...task, owner: store.users[task.ownerId] };
    }),
  };
}
```

Well, this is just a query! If we had a local database — let's call it Local DB — that understood some GraphQL-looking language, we could instead declare:

```
teams {
  ...
```

```
  tasks: {
    ...
    owner: {
      ...
    }
  }
}
```

And voila, we'd have data for our screens.

Next, we backed our data into IndexedDB. Well, databases are good at caching. Our Local DB could back itself up in IndexedDB!

And the mutation system? If our Local DB and Backend DB spoke the same language, both could understand and apply the same mutations. Local DB can handle undo/redo, and with that we have optimistic updates out of the box.

What about sockets? Databases handle replication. So what if we made the client a special node? The Local DB already knows the queries to satisfy. So it can talk to the backend and get the data it needs.

On the backend, what if we had the same kind of permission system that Facebook had? We'd have a fully expressive language that could scale to complex rules.

Make the Backend DB handle live queries, and we have all the pieces for our missing column!

Local DB

Let's dive into our Local DB first. This is what's going to handle queries, caching, and talking to our server. If we do this right, we inform everything else.

Requirements

The minimum our Local DB needs is support for relations. Whatever we do, we should be able to express “Give me team info, related tasks, and the owner for each task”.

We should also support recursive queries. For Notion, we need to say “Give me a block and expand all children recursively”.

Our Local DB should also be easy to use. Firebase is famous for this. You can start working with a single index.html file. API calls are consistent and simple. You don’t need to specify a schema to get started. We should be just as easy to use. [18]

And our Local DB should be light. At least on the client. Yes we can cache the download. But I don’t think developers will take you up on an offer that doubles their bundle.

Finally, our Local DB should be simple. Every feature in our Local DB needs to be supported by our multiplayer backend. This won’t ship if our spec is too large.

Exploring SQL

A SQL-based tool is closest at hand. I enjoyed looking at absurd-sql. This uses sql.js (SQLLite compiled to webassembly) and persists state into IndexedDB.

SQL is battle tested and supports a wide array of features. But if you take the constraints we set out, you’ll see it’s a bad bet.

Schema and Size

My investigation began with two light issues.

First, SQL has a schema. Schema is useful, but it make things less easy than Firebase. You can hack immediately in Firebase, but there’s upfront work with a schema. [19]

Second, there's size. sql.js is about 400KBs gzipped. Yes this can be cached, but I just don't see most apps adopting a library that adds this overhead.

Both reservations have reasonable counters. We could infer a schema on our user's behalf, or write a lighter implementation of SQL. With problems like this we could have moved forward.

Language

But SQL as a language turns out to be a dealbreaker. SQL isn't simple or easy. It's a tough combination of lots of features, with little of it being useful for the frontend.

Consider the most common query for UIs: Fetch nested relations. Remember our `dataForTaskPage` ?

```
function dataForTaskPage(store, teamId) {
  return {
    ...store.teams[teamId],
    tasks: store.tasksForTeam(teamId).map((task) => {
      return { ...task, owner: store.users[task.ownerId] };
    }),
  };
}
```

This is one SQL query for it:

```
SELECT
  teams.*, tasks.*, owner.*
FROM teams
JOIN tasks ON tasks.team_id = teams.id
JOIN users as owner ON tasks.owner_id = owner.id
WHERE teams.id = ?
```

And it works. But it's inconvenient. Our query will return an exploded list of rows. Each row represents an owner, with tasks and teams duplicated. But what we actually wanted was a nested structure. Something like:

```
{
  teams: [{id: 2, name: "Awesome Team", tasks: [{..., owner: {}}, ...]}, ...]
}
```

To make this work, we could use a `GROUP BY` with `json_group_array` and `json_object`. Like this:

```
SELECT
  teams.*,
  json_group_array(
    json_object(
      'id', tasks.id,
      'title', tasks.title,
      'owner', json_object('id', owner.id, 'name', owner.name))
  ) as tasks
FROM teams
JOIN tasks ON tasks.team_id = teams.id
JOIN users as owner ON owner.id = tasks.owner_id
GROUP BY teams.id
WHERE teams.id = ?
```

Try it here.

But you can already see we're going off the beaten path. What if we had subscribers for each task? We'd need at least two more joins. One more `GROUP BY`. Likely we'd want a subquery. And if we wanted to support the Notion case? We'd want a `WITH RECURSIVE` clause.

Now we're in a tough spot. The frontend's common case is SQL's advanced case. We shouldn't need advanced features for common cases.

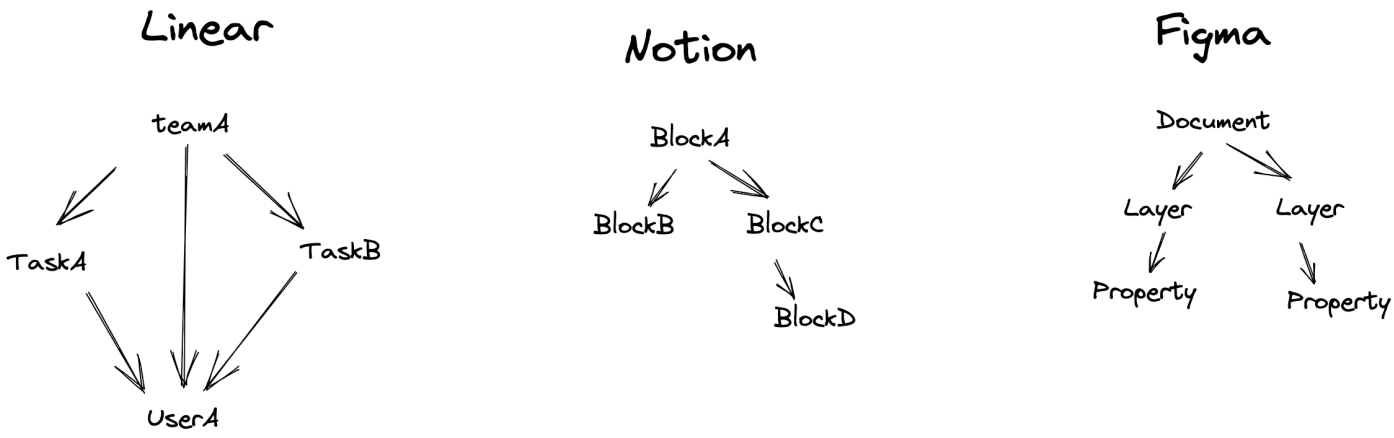
Plus, what about all the SQL features we'd rarely use in the frontend? The spec for the core language is over 1700 pages long [20]. We'd have to implement reactivity for all 1700 pages. I don't think the schlep is worth it.

Another Approach

SQL is out. Let's start with a different question then: How do we make frontend queries easy?

The most common query is our "fetch nested relations". For Linear it's "team, with related tasks and their owners". Or for Notion, we want "blocks, with child blocks expanded". Or for Figma, "documents with their comments, layers, and properties".

See a pattern here? They're all graphs:



And this pointed us to a question: would a graph database make frontend queries easy?

Triple Stores

So we wrote a graph database to find out. We chose Triple Stores, one of the simplest kinds of graph databases. If you haven't tried one, here's a quick intuition:

Imagine we're trying to express a graph with data structures. What do we need?

Well, we need to be able to express a node with attributes. To say:

```
User with id 1 has name "Joe"
Team with id 2 has name "Awesome Team"
Task with id 3 has title "Code"
```

These sentences translate to lists:

```
[1, "name", "Joe"]
[2, "name", "Awesome Team"]
[3, "title", "Code"]
```

Then we want a way to describe references. To say:

```
Task with id 3 has an "owner" reference to User with id 1
Team with id 2 has a "task" reference to Task with id 3
```

Well...these translate to lists just as well:

```
[3, "owner", 1]
[2, "tasks", 3]
```

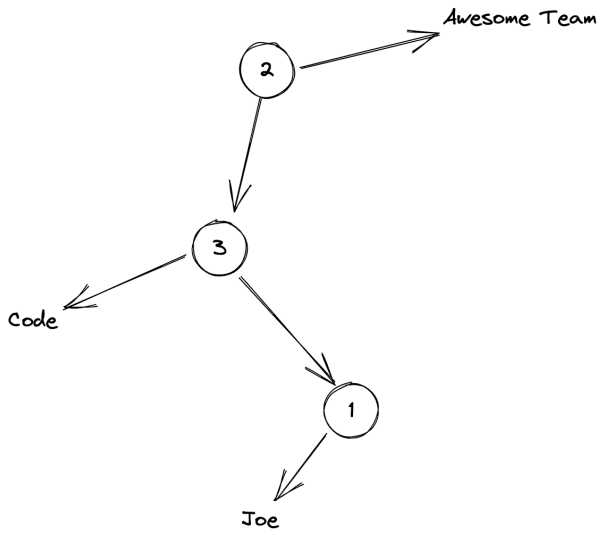
Put these lists in a table, and you have a triple store! *Triple* is the name of the list we've been writing:

```
[1, "name", "Joe"]
```

The first item is always an `id`, the second the `attribute`, and the third, the `value`. Turns out triples are all we need to express a graph. Here's a more fleshed out example:

Triple Stores

id	attribute	value
1	user/name	Joe
2	team/name	Awesome Team
3	task/title	Code
2	team/task	3
3	task/owner	1



And once you've expressed a graph, you can traverse it. Triple stores have interesting query languages. Here's Datalog:

```
(pull db '[* {:team/task [* {:task/owner [*]}]}] team-id)
```

With this we've replaced `dataForTasksPage` !

Exploring Triple Stores

Triple stores felt like our rubicon moment. An entire architecture unravelled from our choice.

Schema and Size

My investigation kicked off with two happy surprises.

First, I always assumed that if we wanted relations, we would need a schema. But it turns out triple stores don't need one. [21] I think a schema is helpful. But to compete with Firebase, it's a win that we can make this optional.

Then there's size. Triple stores are notoriously light. Datascript is one of the most battle-tested triple stores. It's transpiled from Clojurescript and carries the extra weight of Clojure. But even then, the bundle size is about 90KB.

Simple

But the killer feature is how simple triple stores are. **You can write a roughly complete implementation in less than a hundred lines of Javascript** [22].

The query planner uses 3 main indexes [23]. Datalog — the query language I mentioned — is so simple that there isn't a spec [24]. The mutation system boils down two primitives [25].

Even with the 100 LOC version, you can express a query like "Give me all the owners for the tasks where this person is a subscriber" [26]

80/20 for Multiplayer

Turns out triple stores are a great answer for muliplayer too. Once we make our Local DB collaborative, we'll need to support conflicts. What should happen when two people change something at the same time?

Notion, Figma, and Linear all use last-write-wins. This means that whichever change reaches the server last wins.

This can work well, but we need to be creative about it. Imagine if two of us changed the same Figma Layer. One of us changed the font size, and the other changed the background color. If we're creative about how we save things, there shouldn't be a conflict in the first place.

How does Figma this? They store their properties in a special way. They store them as...triples! [27]

```
[1, "fontSize", 20]
[1, "backgroundColor", "blue"]
```

These triples say that the Layer with id 1 has a fontSize 20 and backgroundColor blue. Since they are different rows, there's no conflict.

And voila, we have the same kind of conflict-resolution as Figma. [28]

But Speed and Scale?

At this point, you may wonder: this is great and all, but what about speed and scale?

Well, the core technology is old [29]. Datalog and triple stores have been around for decades. This also means that people have built reactive implementations [30].

But what makes me most optimistic about the answer here, is that Facebook runs on a graph database. Tao is facebook's in-house data store. If you look at Tao, it's not so different from a triple store! [31]

Easy?

This is getting exciting. But what about ease of use? This is how the "Give me all the owners for the tasks where this person is a subscriber" query looks in Datalog:

```
{:find ?owner,
  :where [[?task :task/owner ?owner]
          [?task :task/subscriber sub-id]}
```

Datalog as a language is elegant and simple. But it's not easy the same way Firebase is. You need to learn a logic-based language. Then you get back triples. But in the UI you want typed objects.

This would be a deal-breaker. But here's where Datalog's strength comes in. **It's so small that we can just keep it as our base layer, and write a friendlier language on top.**

InstaQL

That's how InstaQL was born. If you look at what's intuitive for the UI, I think GraphQL syntax comes closest:

```
teams {
  ...
  tasks: {
    ...
    owner: {
      ...
    }
  }
}
```

You just declare what you want; the shape of the query looks like the result.

InstaQL was heavily inspired by GraphQL. It's a similar-looking language and produces Datalog. Here's how queries look:

```
{
  teams: {
    $: {where: {id: 1}},
    tasks: {owner: {}},
  },
}
```

You can see the first departure from GraphQL: InstaQL is written with plain javascript objects. This lets us avoid a build step; after all Firebase doesn't need one. And there's another win: if the language itself is written with objects and arrays, engineers can write functions that manipulate them.

The second departure is in the mutation system. In GraphQL you define mutations as functions in the backend. This is a problem because then you can't do optimistic updates out of the box. Without talking to the server, there's no way to know what a mutation does.

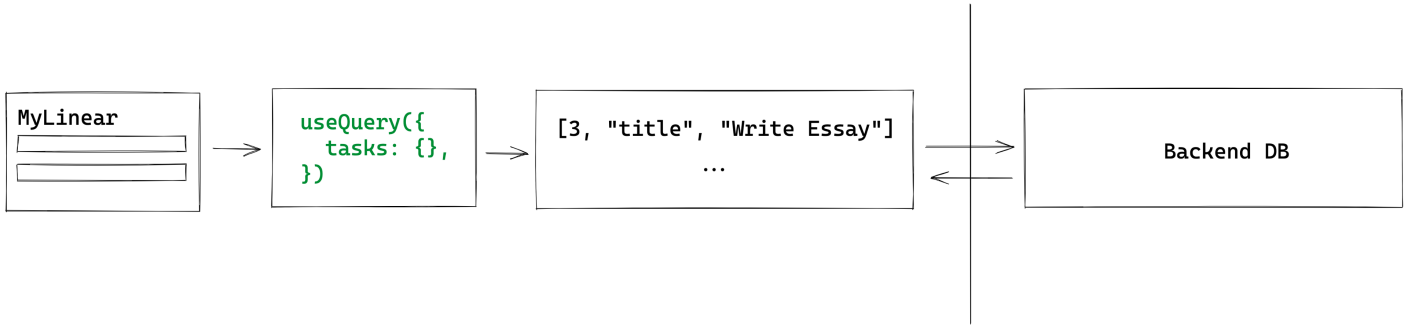
In InstaQL, mutations look like this:

```
transact([
  tx.tasks[taskId]
    .update({title: "New Task"})
    .link({owner: ownerId}}
])
```

These mutations produce triple store assertions and retractions. So our Local DB can apply them, and we have optimistic updates out of the box again. [32]

Instant Today

So we wrote a triple store, and Instant was born. Here's roughly how it looks:



We have a client-side implementation of InstaQL. You can write queries like:

```
{
  teams: {
    $: {where: {id: 1}},
    tasks: {owner: {}}
  },
}
```

And get back objects:

```
{
  teams: [{
    id: 1,
```

```
    name: 'Awesome Team',
    tasks: [{id: 3, title: 'Code', owner: [{id: 1, name: 'Joe'}]
  }]
}
```

These are live queries that talk to the Local DB — a triple store. The Local DB then handles optimistic updates and syncs with the backend server, sockets and all [33].

When our early users wrote their first relational query, I saw delight in their eyes. And boy was that thrilling.

Instant Tomorrow

Right now we support email-only auth. But we don't have permission system yet. Either data is anonymous and can be written by everyone, or users are logged in and can only read and write their own data.

This is limiting, but it's in a state where you can play. Coming soon, we're going to build an FB-like permission system. This way you can write expressive permissions like this:

```
function IDenyIfArchived(_user, task) {
  if (task.isArchived) {
    return deny();
  }
  return allow();
}
// ...
{
  "task": {
    read: [
      IAllowIfTeamUser,
    ],
    write: [
      IDenyIfArchived,
      IAllowIfTeamUser,
    ],
```

```
  }
}
```

Once we have permissions, we'll be able to make sync more sophisticated. Right now Instant fetches the world. Shortly, Instant will fetch just the queries you need. This will start out simple, but will evolve towards incremental view maintenance.

It was obvious to me that the browser has been missing a database. With Instant, I see that a graph database is one of the best bets we have to make delightful apps easy to build.

If you're excited about this stuff, sign up and give us a try. Joe and I will reach out to your personally for feedback.

Thanks Joe Averbukh, Alex Reichert, Mark Shlick, Slava Akhmechet, Nicole Garcia Fischer, Daniel Woelfel, Jake Teton-Landis, Rudi Chen, Dan Vingo, Dennis Heihoff for reviewing drafts of this essay.

Thoughts? Reach out to me via twitter or email :)