Published July 1, 2024 in Tech

Building and scaling Notion's data lake

By XZ Tie, Nathan Louie, Thomas Chow, Darin Im, Abhishek Modi, Wendy Jiao

12 min read

In the past three years Notion's data has expanded 10x due to user and content growth, with a doubling rate of 6-12 months. Managing this rapid growth while meeting the ever-increasing data demands of critical product and analytics use cases, especially our recent Notion AI features, meant building and scaling Notion's data lake. Here's how we did it.

Notion's data model and growth

Store tables as a block

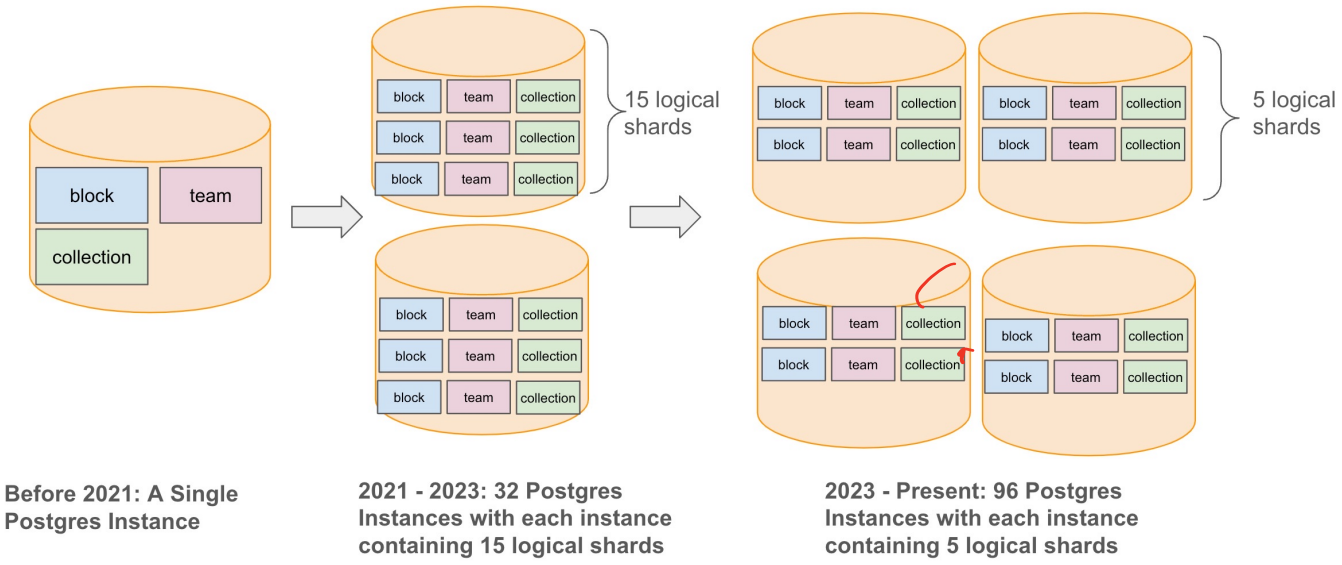
Everything you see in Notion—texts, images, headings, lists, database rows, pages, etc—despite differing front-end representations and behaviors, is modeled as a "block" entity in the back end and stored in the Postgres

database with a consistent structure, schema, and associated metadata (learn more about Notion's data model).

Everything in Notion is a block, and these blocks are made up of data. Lots and lots of data.

All this block data has been doubling every 6 to 12 months, driven by user activity and content creation. At the start of 2021 we had more than 20 billion block rows in Postgres, and this figure has since grown to more than two hundred billion blocks—a data volume of hundreds of terabytes, even when compressed.

To manage this data growth while enhancing the user experience, we've strategically expanded our database infrastructure from one Postgres instance to a more complex sharded architecture. We began in 2021 by horizontally sharding our Postgres database into 32 physical instances, each comprising 15 logical shards, and continued in 2023 by increasing the number of physical instances to 96, with five logical shards per instance. Thus we maintained a total of 480 logical shards while ensuring long-term scalable data management and retrieval capabilities.

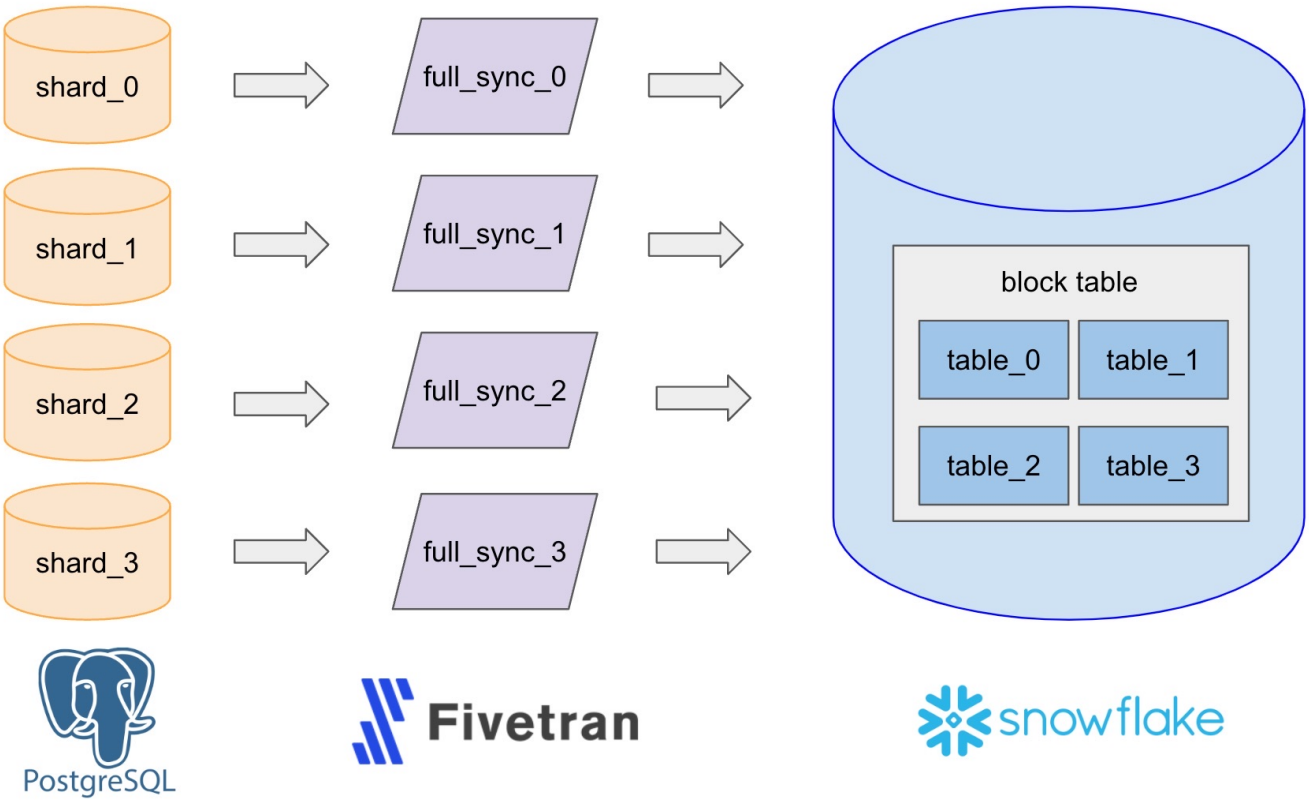


By 2021 Postgres comprised the core of our production infrastructure, handling everything from online user traffic to various offline data analytics and machine learning needs. As the demands on both online and offline data increased, we realized it was essential to build a dedicated data infrastructure to handle offline data without interfering with online traffic.

Notion's data warehouse architecture in 2021

In 2021, we initiated this dedicated data infrastructure with a simple ELT (Extract, Load, and Transform) pipeline that used the third-party tool Fivetran to ingest data from the Postgres WAL (Write Ahead Log) to Snowflake and set up 480 hourly-run connectors for the 480 shards to write to the same number of raw Snowflake tables. We then merged these tables into a single large table for analytics, reporting, and machine learning use cases.

Must be similar to atomic. events in Snowplow



Scaling challenges

As our Postgres data grew, we encountered several scaling challenges.

Operability

The overhead of monitoring and managing 480 Fivetran connectors, along with re-syncing them during Postgres re-sharding, upgrade, and maintenance periods, became extremely high, creating a significant on-call burden for team members.

Speed, data freshness and cost

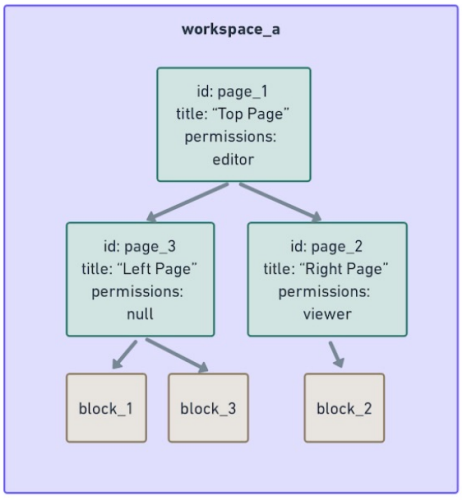
Ingesting data to Snowflake became slower and more costly, primarily due to Notion's unique update-heavy workload. Notion users update existing blocks (texts, headings, titles, bullet lists, database rows, etc) much more often than they add new ones. This causes block data to be predominantly update-heavy —90% of Notion upserts are updates. Most data warehouses, including

Snowflake, are optimized for insert-heavy workloads, which makes it increasingly challenging for them to ingest block data.

Use case support

Data transformation logic became more complex and heavy, surpassing the capabilities of the standard SQL interface offered by off-the-shelf data warehouses.

- One important use case is constructing denormalized views of Notion's block data for key products (e.g., AI and Search). Permission data, for example, ensures that only the right people can read or change a block (this blog discusses Notion's block permission model). But a block's permission isn't statically stored in the associated Postgres—it has to be constructed on the fly via expensive tree traversal computation.
- In the following example, `block_1`, `block_2`, and `block_3` inherit permissions from their immediate parents (`page_3` and `page_2`) and ancestors (`page_1` and `workspace_a`). To build permission data for each of these blocks, we must traverse its ancestor tree all the way up to the root (`workspace_a`) in order to ensure completeness. With hundreds of billions of blocks whose ancestor depths ranged from a few to dozens, this kind of compute was very costly and would simply time out in Snowflake.



id	ancestors	permissions	path_text	depth
page_1	[workspace_a]	editor	"Top Page"	1
page_2	[page_1, workspace_a]	viewer	["Top Page", "Right Page"]	2
page_3	[page_1, workspace_a]	editor	["Top Page", "Left Page"]	2
block_1	[page_3, page_1, workspace_a]	editor		3
block_2	[page_2, page_1, workspace_a]	viewer		3
block_3	[page_3, page_1, workspace_a]	editor		3

Constructing permission data for each block requires traversing the entire ancestor tree. ⤢

Because of these challenges, we started to explore building our data lake.

Building and scaling Notion's in-house data lake

Here were our objectives for building an in-house data lake:

- Establish a data repository capable of storing both raw and processed data at scale.
- Enable fast, scalable, operable, and cost-efficient data ingestion and computation for any workload—especially Notion's update-heavy block data.
- Unlock AI, Search, and other product use cases that require denormalized data.

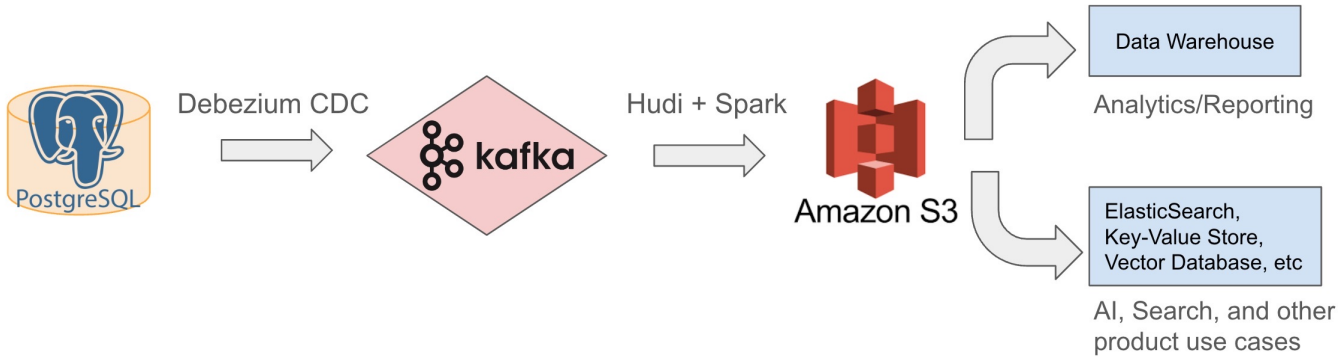
However, while our data lake is a big step forward, it's important to clarify what it's not intended to do:

- Completely replace Snowflake. We'll continue to benefit from Snowflake's operational and ecosystem ease by using it for most other workloads, particularly those that are insert-heavy and don't require large-scale denormalized tree traversal.
- Completely replace Fivetran. We'll continue taking advantage of Fivetran's effectiveness with non-update heavy tables, small dataset ingestion, and diverse third-party data sources and destinations.
- Support online use cases that require second-level or stricter latency. The Notion data lake will primarily focus on offline workloads that can tolerate minutes to hours of latency.

Real- time usecases still rely on other data stores

Our data lake's high-level design

Since 2022 we've used the in-house data lake architecture shown below. We ingest incrementally updated data from Postgres to Kafka using Debezium CDC connectors, then use Apache Hudi, an open-source data processing and storage framework, to write these updates from Kafka to S3. With this raw data we can then do transformation, denormalization (e.g., tree traversal and permission data construction for each block), and enrichment, then store the processed data in S3 again or in downstream systems to serve analytics and reporting needs, as well as AI, Search, and other product requirements.



Notion's in-house data lake is built on Debezium CDC connector, Kafka, Hudi, Spark, and S3.

Next we'll describe and illustrate the design principles and decisions we arrived at after extensive research, discussion, and prototyping work.

Design decision 1: Choosing a data repository and lake

Our first decision was to use S3 as a data repository and lake to store all raw and processed data, and position data warehouse and other product-facing data stores such as ElasticSearch, Vector Database, Key-Value store, etc as its downstream. We made this decision for two reasons:

- It aligned with Notion's AWS tech stack, e.g., our Postgres database is based on AWS RDS and its export-to-S3 feature (described in later sections)

allows us to easily bootstrap tables in S3.

- S3 has proven its ability to store large amounts of data and support various data processing engines (like Spark) at low cost.

By offloading heavy ingestion and compute workloads to S3 and only ingesting highly cleaned and business-critical data to Snowflake and product-facing data stores, we significantly improved data compute scalability and speed and reduced cost.

This is when snowflake mandated storage on ons's end. Charged with Iceberg

Design decision 2: Choosing our processing engine

We chose Spark as our main data processing engine because as an open-source framework, it could be rapidly set up and evaluated to verify that it met our data transformation needs. Spark has four key benefits:

No justification YET! :)

- Spark's wide range of built-in functions and UDFs (User Defined Functions) beyond SQL enable complex data processing logics like tree traversal and block data denormalization, as described above.
- It offers a user-friendly PySpark framework for most lighter use cases, and advanced Scala Spark for high-performance, heavy data processing.
- It processes large-scale data (e.g., billions of blocks and hundreds of terabytes) in a distributed manner, and exposes extensive configurations, which allows us to fine-tune our control over partitioning, data skewness, and resource allocation. It also enables us to break down complex jobs into smaller tasks and optimize resourcing for each task, which helps us achieve reasonable runtime without over-provisioning or wasting resources.
- Finally, Spark's open-source nature offers cost-efficiency benefits.

Design decision 3: Preference for incremental ingestion over snapshot dump

After finalizing our data lake storage and processing engine, we explored

solutions for ingesting Postgres data to S3. We wound up considering two approaches: incremental ingestion of changed data and periodic full snapshots of Postgres tables. In the end, based on performance and cost comparisons, we opted for a hybrid design:

- **During normal operations, incrementally ingest and continuously apply changed Postgres data to S3.**
- **In rare cases, take a full Postgres snapshot once to bootstrap tables in S3.**

The incremental approach ensures fresher data at lower cost and with minimal delay (a few minutes to a couple hours, depending on table size). Taking a full snapshot and dumping to S3, by contrast, takes more than 10 hours and costs twice as much, so we do so infrequently, when bootstraping new tables in S3.

Design decision 4: Streamlining incremental ingestion

- **Kafka CDC Connector for Postgres → to → Kafka**

We chose the Kafka Debezium CDC (Change Data Capture) connector to publish incrementally changed Postgres data to Kafka, similar to Fivetran's data ingestion method. We chose it together with Kafka for their scalability, ease of setup, and close integration with our existing infrastructure.

- **Hudi for Kafka → to → S3**

Hudi is good for update heavy workload

To ingest the incremental data from Kafka to S3, we considered three excellent data lake and ingestion solutions: Apache Hudi, Apache Iceberg, and DataBricks Delta Lake. In the end we chose Hudi for its excellent performance with our update-heavy workload and its open-source nature and native integration with Debezium CDC messages.

Iceberg and Delta Lake, on the other hand, weren't optimized for our update-

heavy workload when we considered them in 2022. Iceberg also lacked an out-of-box solution that understands Debezium messages; Delta Lake does have one, but it isn't open source. We would have had to implement our own Debezium consumer if we'd gone with either of those solutions.

Design decision 5: Ingest raw data before processing

Finally, we decided to ingest raw Postgres data to S3 without on-the-fly processing in order to establish a single source of truth and simplify debugging across the entire data pipeline. Once raw data is in S3, we then do transformation, denormalization, enrichment, and other types of data processing. We store intermediate data in S3 again and only ingest highly cleaned, structured, and business-critical data to downstream systems for analytics, reporting, and product needs.

store in S3 & only process from S3

Scaling and operating our data lake

We experimented with many detailed setups in order to tackle the scalability challenges associated with Notion's ever-increasing data volume. Here's what we tried and how it went:

1. CDC connector and Kafka setup

We set up one Debezium CDC connector per Postgres host and deploy them in an AWS EKS cluster. Because of the maturity of Debezium and EKS management and Kafka's scalability, we've only had to upgrade the EKS and Kafka clusters a few times in the past two years. As of May 2024, it smoothly handles tens of MB/sec of Postgres row changes.

We also configure one Kafka topic per Postgres table and let all connectors consuming from 480 shards write to the same topic for that table. This setup

significantly reduced the complexity of maintaining 480 topics for each table and simplified downstream Hudi ingestion to S3, significantly reducing operational overhead.

2. Hudi setup

We used Apache Hudi Deltastreamer, a Spark-based ingestion job, to consume Kafka messages and replicate the state of Postgres table in S3. After several rounds of performance tuning, we established a fast, scalable ingestion setup to ensure data freshness. This setup provides a delay of just a few minutes for most tables, and up to two hours for the largest one, the block table (see graphic below).

- We use the default COPY_ON_WRITE Hudi table type with UPSERT operation, which suits our update-heavy workload.
- To manage data more effectively and minimize write amplification (i.e., the number of files updated per batched ingestion run), we fine-tuned three configurations:
  - Partition/shard data using the same Postgres shard scheme, i.e., the `hoodie.datasource.write.partitionpath.field: db_schema_source_partition` config. This partitions the S3 dataset into 480 shards, from `shard0001` to `shard0480`, making it more likely that a batch of incoming updates map to the same set of files from the same shard.
  - Sort data based on the last updated time (event_lsn), i.e., the `source-ordering-field: event_lsn` config. This is based on our observation that more recent blocks are more likely to get updated, which allows us to prune files with only outdated blocks.
  - Set the index type to be bloom filter, i.e., the `hoodie.index.type: BLOOM` config, to further optimize the workload.

Hudi Deltastreamer setup for the block table.

3. Spark data processing setup

For the majority of our data processing jobs we utilize PySpark, whose relatively low learning curve makes it accessible to many team members. For more complex jobs such as tree traversal and denormalization, we leverage Spark's superior performance in several key areas:

- We benefit from the performance efficiency of the Scala Spark.
- We manage data more effectively by handling large and small shards separately (remember we kept the same 480 shards scheme in S3 to be consistent with Postgres); small shards have their entire data loaded into the Spark task container memory for fast processing, whereas large shards that exceed memory capacity are managed through disk reshuffling.
- We utilize multi-threading and parallel processing to speed up processing of 480 shards, allowing us to optimize runtime and efficiency.

4. Bootstrap setup

Here's how we bootstrap new tables:

- We first set up Debezium Connector to ingest Postgres changes to Kafka.
- Starting from timestamp `t`, we kick off a AWS RDS-provided export-to-S3 job to save the latest snapshot of Postgres tables to S3. We then create a Spark job to read those data from S3 and write them to the Hudi table format.
- Finally, we ensure that all changes made during the snapshotting process are captured by setting up Deltastreamer to read from Kafka messages from `t`. This step is crucial for maintaining data completeness and integrity.

Thanks to the scalability of Spark and Hudi, these three steps usually complete within 24 hours, allowing us to perform re-bootstrap with manageable time to accommodate new table asks and Postgres upgrade and re-sharding operations.

The payoff: less money, more time, stronger infrastructure for AI

We started developing our data lake infrastructure in the spring of 2022 and completed it by that fall. Due to the infra's inherently scalable nature, we were able to continually optimize and expand the Debezium EKS clusters, Kafka clusters, Deltastreamer, and Spark job to keep up with Notion's 6-to-12 month data doubling rate without significant overhauls. The payoff was significant:

- Moving several large, crucial Postgres datasets (some of them tens of TB large) to data lake gave us a net savings of over a million dollars for 2022 and proportionally higher savings in 2023 and 2024.
- For these datasets, the end-to-end ingestion time from Postgres to S3 and

Snowflake decreased from more than a day to a few minutes for small tables and up to a couple of hours for large ones. Re-syncs, when necessary, can be completed within 24 hours without overloading live databases.

- Most importantly, the changeover unlocked massive data storage, compute, and freshness savings from a variety of analytics and product asks, enabling the successful rollout of Notion AI features in 2023 and 2024. Stay tuned for a detailed post on our Search and AI Embedding RAG Infra built on top of the data lake!

We'd like to thank OneHouse and the Hudi open source community for their tremendous and timely support. Great open source support was crucial to our ability to spin up the data lake in just a few months.

As our needs grow and diversify, we continue to enhance our data lake by building automated and self-service frameworks to empower more engineers to manage and develop product use cases based on the data.

Interested in helping us build the next generation of Notion's data management? Apply for our open roles here.

Keep reading

All posts →

Notion HQ

Four key elements to building an AI team

Tech

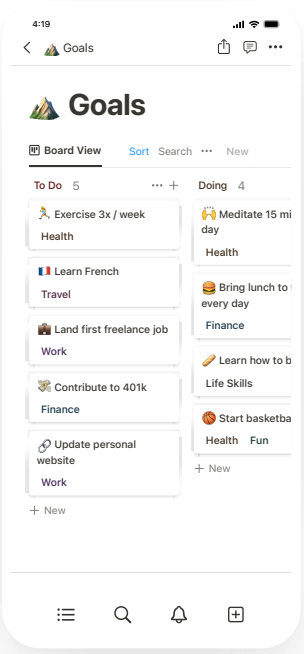
Gen Z is changing the way we work. Are companies prepared?

Inspiration

The best student and teacher templates to prep for day one of school

Try it now

We also have Mac & Windows apps to match.



Download from App Store

Notion

English

Do Not Sell or Share My Info

Cookie settings

Company

About us

Careers

Security

Status

Terms & privacy

Download

iOS & Android

Mac & Windows

Calendar

Web Clipper

Resources

Help center

Pricing

Blog

Community

Integrations

Templates

Affiliates

Notion for

Enterprise

Small business

Personal

Explore more →