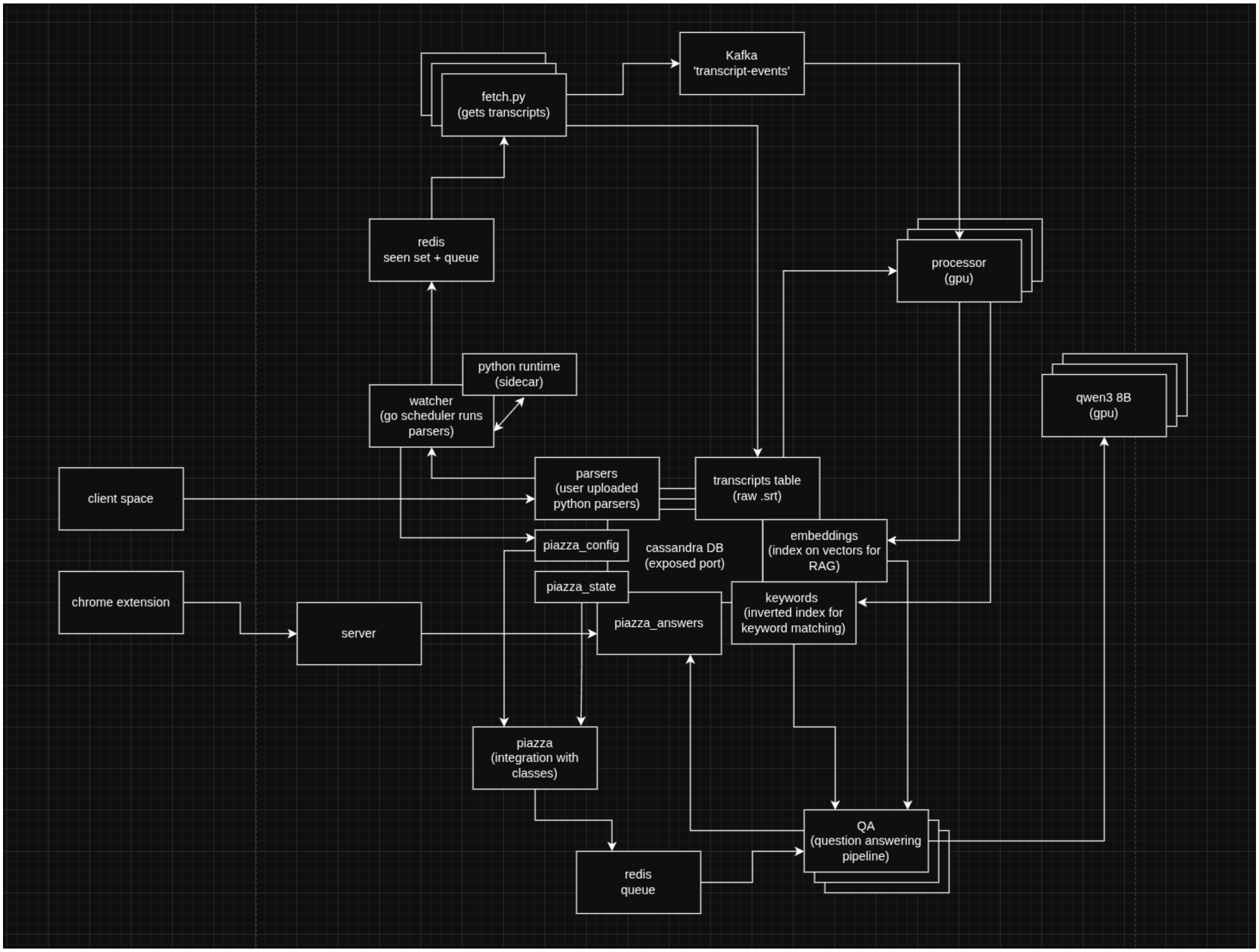
Piazza Bot

Quick Start

1. Clone the repository
    - git clone git@github.com:njyeung/piazza-bot.git
    - cd piazza-bot
2. Implement parsers
    - Since each professor has their own unique website, if a user wishes to scrape a website that does not already have a parser written for it, they must write a parser to print kaltura gallery links obtained from that website using the format: https://mediaspace.wisc.edu/media/<lecture title>/<entry_id>
    - This parser will be a python file placed in /piazza-bot/parsers/
    - Refer to cs544.py as an example
    - To upload the new parsers to the cassandra database, the user must run "python manage.py apply"
3. Modify environment variables in `docker-compose.yml if you'd like`:
    - `CASSANDRA_HOSTS` - Cassandra cluster nodes
    - `CASSANDRA_KEYSPACE` - Database keyspace name
    - `REDIS_HOST`, `REDIS_PORT` - Redis connection
    - `REDIS_QUEUE` - Job queue name
    - `REDIS_SEEN_SET` - Set for tracking processed URLs
4. Start the cluster using
    - ./build.sh
    - docker-compose up
    - Access logs using ```bash docker-compose logs -f fetcher``` and ```bash docker-compose logs -f watcher```
    - After bringing down the cluster, run ./build.sh again before doing docker compose up otherwise it will break



Client space

Since Cassandra has an exposed port, the user can interact directly with the database to upload their parsers. A parser uses Selenium to scrape a course website for lecture links. When it is run, it should take no arguments and print out json objects to stdout.

```
> python cs544.py
{"class_name": "CS544", "professor": "Tyler", "semester": "FALL25", "url": "<https://mediaspace.wisc.edu/media/…>", "lecture_title": "Memory Resources (Caching)"}
{"class_name": "CS544", "professor": "Tyler", "semester": "FALL25", "url": "<https://mediaspace.wisc.edu/media/…>", "lecture_title": "Memory Resources (Caching Practice)"}
…
```

Parsers should be placed in the /parsers directory. There is already a cs544.py parser that scrapes Tyler's course website.

To sync parsers with the Cassandra database:
- cd piazza-bot
- python -m venv venv
- source venv/bin/activate
- pip install -r requirements.txt
- python manage.py apply

This operation is idempotent. Whatever is currently in the parsers directory will be synced with the database. If a parser was previously in /parsers but was removed, the database will also delete that parser.

Watcher (/watcher Go)

The watcher polls the database and runs parsers on a scheduled loop (once every day but currently once every minute for debugging purposes). It forwards the user's parser output into a redis queue, and adds each url to the seen set.

It is also responsible for uploading the parser headers (containing piazza network id and credentials) into the piazza_config table.

Workers (fetch.py)

Fetches transcripts (.srt) from kaltura gallery using Selenium.

It inserts each raw transcript into a Cassandra table along with metadata.

```
// table schema from init_db.py
CREATE TABLE IF NOT EXISTS transcripts (
    class_name text,
    professor text,
    semester text,
    url text,
    lecture_number int,
    lecture_title text,
    transcript_text text,
    downloaded_at timestamp,
    status text,
    PRIMARY KEY ((class_name, professor, semester), url)
)
```

The web scrapers are high throughput, they insert with ConsistencyLevel.ONE. We want to maintain R+W > RF. Since our replication factor is 3 (db-1, db-2, db-3), the heavy *processor* step later in the pipeline has a W=3.

After inserting into the transcripts table, the fetcher also sends a message through the Kafka stream:

```
// from fetch.py
event = {
        "class_name": lecture.get("class_name"),
        "professor": lecture.get("professor"),
        "semester": lecture.get("semester"),
        "url": url,
        "lecture_number": lecture.get("lecture_number"),
        "lecture_title": lecture.get("lecture_title")
}
```

Processor (/processor Go)

After receiving a Kafka message from the previous step, the processor reads from the corresponding raw transcript row from Cassandra. The processor is responsible for converting raw transcripts into ~500 token chunks that will be used for retrieval augmented generation. After the chunks have been created, it inserts them into the embeddings table:

```
// from init_db.py
CREATE TABLE IF NOT EXISTS embeddings (
        class_name text,
        professor text,
        semester text,
        url text,
        chunk_index int,
        chunk_text text,
        embedding VECTOR<FLOAT, 1024>,        // may change if you brought your own embedding model
        token_count int,
        lecture_title text,
        lecture_timestamp text,
        created_at timestamp,
        PRIMARY KEY ((class_name, professor, semester), url, chunk_index)
)
```

The table is indexed on the **embedding** (for RAG)

```
// from init_db.py
CREATE INDEX IF NOT EXISTS embedding_idx
ON embeddings(embedding)
USING 'SAI'
```

Later, we also allow Qwen to look up chunks by **keywords**. To accommodate this, we create an inverted index on the words in each chunk. For each chunk, we deduplicate the words within that chunk (find a set of unique words in the chunk), then for each word we insert it, as well as its corresponding chunk index and partition key in the embeddings table.

```
CREATE TABLE IF NOT EXISTS keywords (
        term text,
        class_name text,
        professor text,
        semester text,
        url text,
        chunk_index int,
        PRIMARY KEY ((term), class_name, professor, semester, url, chunk_index)
)
```

I spent a decent amount of time on the semantic chunking algorithm that relies on a local tokenizer and a plug-and-play local embedding model. I have published it as a separate standalone server here: https://github.com/njyeung/go-semantic-chunking. The algorithm is described in detail below:

Semantic Chunking Algorithm:

Tuneable parameters:

```
// ChunkingConfig holds all tunable parameters for the semantic chunking algorithm
type ChunkingConfig struct {
        OptimalSize  int     // optimal chunk size (in tokens), no penalty below this (default: 470)
        MaxSize      int     // chunk size hard limit, infinite penalty at or above (default: 512)
        LambdaSize   float32 // Max penalty in "edge units" at MaxSize (default: 3.0)
        ChunkPenalty float32 // Initial penalty per chunk to discourage small chunks (default: 1.0)
}
```

We first preprocess the raw transcripts by removing timestamps. The cleaned text is then segmented into sentences using standard sentence delimiters (. , ? , !).

- Note: To ensure compatibility with downstream embedding and retrieval models, we enforce a hard maximum token limit (*MaxSize*) on all segments. In rare cases where a single sentence exceeds *MaxSize*, we greedily split the sentence into consecutive chunks of length *MaxSize*, with the final chunk containing the remaining tokens. These chunks may end mid-sentence, however, this is acceptable in our case. After this step, all sentences are guaranteed to satisfy TokenCount ≤ MaxSize.

Problem Formation:

Given a sequence of embedded sentences

s0, s1, s2 … s(n-1)

The goal is to partition them into contiguous chunks such that:

- Semantically coherent sentences are grouped together (Cosine similarity)
- Chunk sizes remain close to a target length (OptimalSize)
- No chunk exceeds the hard limit (MaxSize)
- Excessive fragmentation into many small chunks is discouraged (ChunkPenalty)

We will solve this segmentation problem via dynamic programming.

Sentence Similarity Precomputation:

- Each sentence is embedded exactly once.
- For each adjacent pair of sentences (s_i, s_{i+1}), we compute the cosine similarity:
  - sim[i] = cos(embed(s), embed(s+1))
- All rewards are min-max scaled to [0, 1] to ensure they're non-negative, there's always reward for merging sentences, and that higher similarity always increases the desirability of merging sentences.

Optimizations for O(1) scoring in DP:

- A prefix_sim array is also computed, where prefix_sim[k] = sim[0] + sim[1]... sim[k]. This allows for efficient computation of the subsequence *reward(i, j)*.
- Similarly, a prefix_token array is created, which is used in *sizePenalty(i, j)*

DP

We define a DP array where:

- dp[j] = best achievable score when chunking sentences [0,j)
- dp[0] = 0 (when there are 0 sentence, the best score is naturally 0)

Recurrence Relation:

dp[j] = max{ i < j } (dp[i] + reward(i, j) - sizePenalty(i, j) - chunkPenalty)

where:

- *reward* favors grouping semantically similar sentences (always positive)
- *sizePenalty* increases smoothly as token count approaches *MaxSize*, and becomes infinite if the limit is exceeded
- *chunkPenalty* is a constant that discourages overly fine-grained segmentation (creating a new chunk requires enough sentences to overcome the starting penalty)

Additional notes:

- Only segments with a total token count ≤ MaxSize are considered legal.

- We also keep track of a *start* array for reconstructing the solution, where start[j] = optimal starting index of the final chunk ending at j.

sizePenalty Function

The *sizePenalty* mentioned above is a hinge-like function parameterized by:

- OptimalSize: no penalty below this threshold
- MaxSize: hard upper bound
- LambdaSize: maximum penalty applied near MaxSize

This encourages chunks to be *OptimalSize*, while still allowing flexibility when semantic coherence warrants bigger chunks.

Reconstruction

Once dp[n] is computed, we reconstruct the optimal segmentation by backtracking through the start[] array from n to 0. Chunks are built in reverse order and then reversed to restore original sequence order.

Each chunk aggregates:

- Sentence text
- Sentence embeddings
- Total token count
- Lecture timestamp

Finally, for each chunk, the paragraph is embedded one final time before inserting into Cassandra.

Notes

You may bring your own embedding model.

- You will need a .onnx file and the corresponding tokenizer.json (placed in /processor directory)
- Currently, we use https://huggingface.co/thenlper/gte-large/tree/main/onnx

If you decide to run this outside of a docker container, you will need to bring your own onnx C++ runtime (cpu or gpu).

- Currently, we use https://github.com/microsoft/onnxruntime/releases/download/v1.23.2/onnxruntime-linux-x64-gpu-1.23.2.tgz
- Refer to Dockerfile.processor for instructions on how to install this

You may bring your own tokenizer.

- Currently, we use https://github.com/sugarme/tokenizer

Qwen3 (\llm)

Experimented with Qwen3 4B and 8B.

QA Pipeline (\qa-worker)

The QA pipeline takes as input the text of a Piazza post along with class metadata (e.g., semester and course name), which is used to filter the retrieval database, and returns a synthesized answer.

Although Qwen3 is less capable than commercial large language models, Piazza questions do not require real-time responses. This allows the system to trade latency for accuracy by performing multiple passes over the data and iteratively refining the result.

First, Qwen is asked to identify if the post is relevant and answerable. Non-answerable posts include complaints about course structure, grading, and announcements.

Next, Qwen identifies a list of keywords that will be used to query the inverted index. For RAG using the embeddings table, the entire post is embedded, so there is no additional work needed from Qwen.

Then, the system retrieves the top 5 chunks from the embeddings and inverted index table. Retrieved chunks from both sources are deduplicated.

Because transcript chunks are sequential segments of lecture recordings, each selected chunk at index i is expanded to include its immediate neighbors (i − 1, i, i + 1) in order to provide additional contextual continuity.

Each expanded chunk group is processed by an independent subagent, all of which receive the same original question prompt. Each subagent determines whether its assigned context is relevant and, if so, produces a concise summary addressing the question.

The resulting summaries, along with associated metadata such as lecture timestamps, are returned to the main Qwen agent. Based on this information, Qwen synthesizes a final answer with citations to the lecture number and timestamp.

Although this pipeline incurs a large latency, it substantially outperforms directly feeding retrieved chunks into Qwen. In practice, the system can safely evaluate up to 10 expanded chunk groups (i − 1, i, i + 1), meaning up to ~15000 tokens, without introducing hallucinations, resulting in significantly higher answer quality.

Piazza-monitor (/piazza-monitor)

The Piazza monitor periodically polls the piazza_config table to detect newly uploaded classes. Once a config has been present for at least 10 minutes—enough time for the pipeline to process a full semester of lecture transcripts—the monitor uses the Python piazza-api to begin scanning the Piazza forums and ingesting posts.

Because Piazza post IDs are monotonically increasing, the monitor also tracks a per-network offset (the last seen post ID) in the piazza_state table.

Server (/api/server.py)

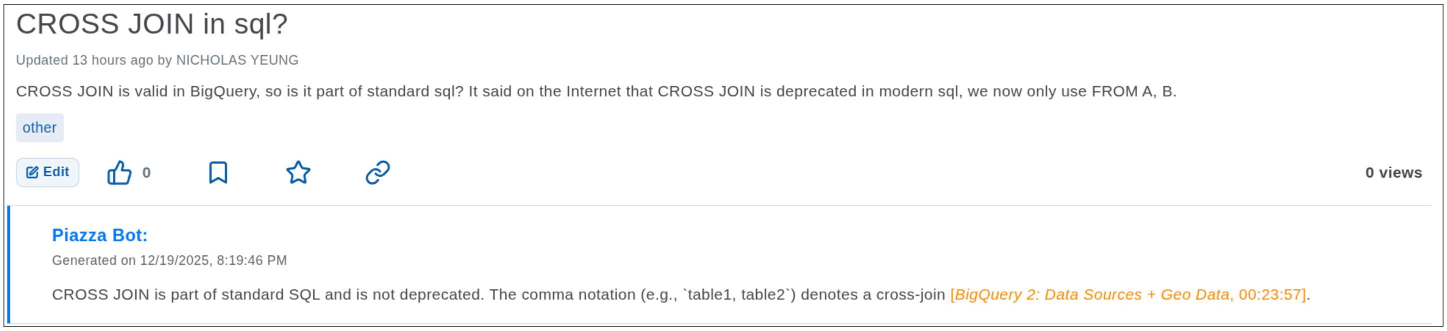
Work in progress server that is simply a wrapper for hitting the Cassandra piazza_answers table.

Chrome Extension

Finally, users can view Piazza Bot's responses directly through a Chrome extension. When an answer exists for a given question in the Cassandra database, the extension displays a small popup beneath the question containing the bot's response. If no answer is available—or if the bot chose not to respond—nothing is displayed.

Performance

Benchmarks on REAL CS544 piazza questions:



CROSS JOIN in sql?

Updated 13 hours ago by NICHOLAS YEUNG

CROSS JOIN is valid in BigQuery, so is it part of standard sql? It said on the Internet that CROSS JOIN is deprecated in modern sql, we now only use FROM A, B.

Piazza Bot:

Generated on 12/19/2025, 8:19:46 PM

CROSS JOIN is part of standard SQL and is not deprecated. The comma notation (e.g., `table1, table2`) denotes a cross-join [*BigQuery 2: Data Sources + Geo Data*, 00:23:57].

join() method

Updated 13 hours ago by NICHOLAS YEUNG

Justo to clarify, does the join method completely stop the thread from executing or does it just pause it?

hw1

Edit 0

0 views

Piazza Bot:

Generated on 12/19/2025, 8:26:14 PM

The join() method completely stops the calling thread from executing until the target thread finishes. [*Compute Resources (Threads)*, 00:07:59] [*Compute Resources (Threads)*, 00:09:57] [*Streaming: Kafka Demos*, 00:41:17] [*BigQuery 4: Cost*, 00:40:06]

S Students' Answer

Where students collectively construct a single answer

Click to start off the wiki answer

0 Followup Discussions

Compose a new followup discussion

Confused about min.insync.replicas behavior

Updated 13 hours ago by NICHOLAS YEUNG

In Meena's lecture today, we learned about the min.insync.replicas property in Kafka. However, I'm still a little confused about the behavior of this property.

It is my understanding that this property only affects thing when the producer is configured to have acks='all'. But my confusion is specifically with how many in-sync replicas need to respond before an ACK to the producer is successful (vs. getting a NotEnoughReplicasException as mentioned in the slides).

If there are currently 5 in-sync replicas (including the leader), and min.insync.replicas is set to 3, then can the producer get an ACK if only 3 of these in-sync replicas respond, or will it still wait until all 5 respond with a successful write?

From what I could tell from doing a little research, the official documentation for Kafka seems to suggest that it will still wait until all currently in-sync replicas respond with a successful write before sending an ACK to a producer that's configured with acks='all' (link to that documentation here), thus in this example, even if 3 of the 5 respond, it will not return an ACK until all 5 respond (or until they stop being in-sync replicas).

Based on this, it seems that min.insync.replicas just acts as a lower barrier to cut off messages from being submitted by producers if not enough in-sync replicas exist at the moment. Is that idea correct?

Thanks.

hw2

Edit 0

0 views

Piazza Bot:

Generated on 12/19/2025, 8:23:12 PM

Your understanding is correct. The `min.insync.replicas` property sets a minimum number of in-sync replicas (including the leader) required for the leader to accept a message. For producers configured with `acks='all'`, the producer receives an ACK only after the message is written to all **currently in-sync replicas** (the current set at the time of the write). Thus, in your example with 5 in-sync replicas and `min.insync.replicas=3`, the producer will not get an ACK until all 5 respond.

[*The Cloud*, 00:00:44]