CS525: Large-Scale Data Management
Project 5
Candidate Ideas

Project 1: Record-Level Indexing

- For each data split, create an inverted index over selected columns
  - Index(es) for each split independently
- At query time
  - Special input format (IF) will be designed
  - IF will accept "trivial" predicates, E.g., column = constant
  - IF will decide which inverted index to use
  - Reads only the records that match the input predicates and pass it to the map task

- **Fits well in one month**
- **Most of the work is in the Input Format and efficient storage for the index**

Project 1: Specifications

- Initial Input: A dataset (say Customers dataset)

- Preprocessing Phase:
  - Design a tool (map-reduce job) that reads each split (split S_i) and creates a corresponding index (D_i)
  - The index will be on a specific column of your choice
    - Theoretically you can create multiple indexes each one is on a different column
  - The index can be an "inverted index", where each line is a value V, and the list of record offsets containing V in the corresponding split

- Query Time:
  - Given a regular job, assume that all selection predicates will be passed to you as parameters in the form of: column_name = constant
  - You design a special input format that should understand the predicates, decide which index to use (if possible), and opens this index to know which records to read from the split
  - If the predicate is on a column not indexed, then the input format will not use the index and will scan all records normally
  - Inside the map function, the normal job will execute (including the predicates again)

- Think about how to search the index fast to be efficient

Project 2: File Tagging

- Add an additional property to files
  - Tag or label
  - A file can have one or more tags

- At query time
  - The job defines a tag, and processes all files having this tag

- **Fits well in one month**
- **Most of the work is in the Input Format and HDFS (NameNode and File objects)**

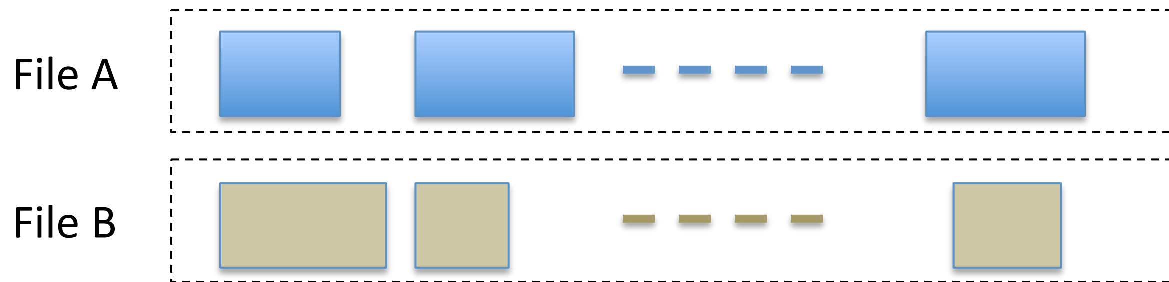
Project 2: Specifications

- Phase I: Adding Tags
  - Investigate the HDFS classes and more specifically the File class and its properties
  - Add new property to each file (Tags), probably as an array of int. So each file can have many tags
  - At the upload time, the file should take additional optional parameter indicating its tags
  - See how the file properties are stored on disk (so when the NameNode restarts it can find this info) and do the same for the new property

- Phase 2: Query Time
  - Instead of specifying a file to read, you will specify a tag (or more) to read in the job
  - Special input format should find all files having this given tag and start reading them as inputs to the job
    - To convert from tags to files, the HDFS should provide a new API (function) to do this job

Project 3: Special Join in Pig

- Pig allows for hints, e.g., "**replicated**" to do broadcast join (for small files)
- What if I want to join A and B, and each are already partitioned on the join key
  - Also a special join (map-only) can be used to do this task
- We need to add a new hint to Pig, e.g., "**partitioned**"
  - Pig now uses special input format to join corresponding partitions



- **Fits well in one month**
- **Most of the work is in understanding Pig's compiler (and try to mimic "replicated" joins)**

Project 3: Specifications

- This one is more challenging than Project 4 (See the next one) for those who want to learn the internals of Pig

- Step 1:
  - Learn how Pig takes a high-level language and converts it to a map-reduce job(s)
    - Focus on simple scenario (E.g., one map-reduce job to join two files)
  - Learn how Pig uses hints like “replicated” keyword to change the implementation of join

- Step 2:
  - Extend Pig by adding a new keyword “partitioned” to implement another type of special joins as shown in the previous slide
  - Try to focus on things that you will change, i.e., try to mimic to a large extent what Pig is doing for “replicated” join.

- Step 3:
  - Compare your new join algorithm with and without the new keyword

Project 4: Performance Comparison (Pig Vs. Java)

- In this project, the internals of Pig will not change
- Find the different types of joins supported by Pig and compare them with "your own" implementation of these joins using Java
- In Java, implement one of the optimized join techniques presented in paper "*A comparison of join algorithms for log processing in mapreduce*"

- **Fits well in one month**
- **Most of the work is in writing java jobs and comparing the performance**

Project 4: Specifications

- Step 1:
  - Find out the different types of Joins supported in Pig and the different scenarios to utilize each one
- Step 2:
  - Implement your own corresponding join jobs using Java
- Step 3:
  - Compare the performance between Pig and Java for the different Join types
- Step 4:
  - Select one optimization from the paper below to implement in Java
  - The paper is: "*A comparison of join algorithms for log processing in mapreduce*"
  - *For example: Instead of reducers caching the records from both relations, with some optimizations, reducers can cache only the records from the smaller relation*
  - *For the optimization you select, discuss whether it can be done in Pig or not*