Chapter 3

Design

The design of Snapstore had two steps. The first was to identify the purposes of a version control system. The six purposes of a version control system identified in [?] were used.

The next step was to create a conceptual model composed of concepts that fulfilled the purposes of a version control system. These concepts were invented in a way that is aligned with conceptual design theory [?]. That is, they were each had a motivating purposes, they were not redundant or overloaded, and they were uniform. A mapping of each Snapstore concept to its motivating purpose is shown in table 3.1. A graphical representation of the conceptual model, using the notation for extended entity-relationship as used in [?], is shown in figure 3-1.

3.1 Data Storage - Snapshot

Persistent data storage in Snapstore is achieved with the notion of a *snapshot*. The snapshot is a saved state of a file. Snapshots record edits, renames, moves, deletes, and merges. Records of the many operations enable all VCS features of Snapstore. The *head* snapshot for any given file is the most recent snapshot made for that file and the current content of that file.

A snapshot is either a create, an update, a rename, or a delete snapshot. This

Concept	**Motivating Purpose**
Snapshot	Making a set of changes to a file persistent
Create, Update, Rename, and Delete Snapshot	Track various types of changes to a file -> making a set of changes to a file persistent
Snapstore Folder	Create, read, and update files -> making a set of changes to a file persistent
Tracked File	Mark files whose changes should be saved -> making a set of changes to a file persistent
Untracked File	Mark files whose changes should be ignored -> making a set of changes to a file persistent
Group	Grouping logically related changes together
Tag	Represent and record coherent points in history
Upstream Repository	Synchronize changes of collaborators
Branch	Support parallel lines of work
Local Repository	Do work in disconnected mode

Table 3.1: VCS purpose to concept mapping for Snapstore

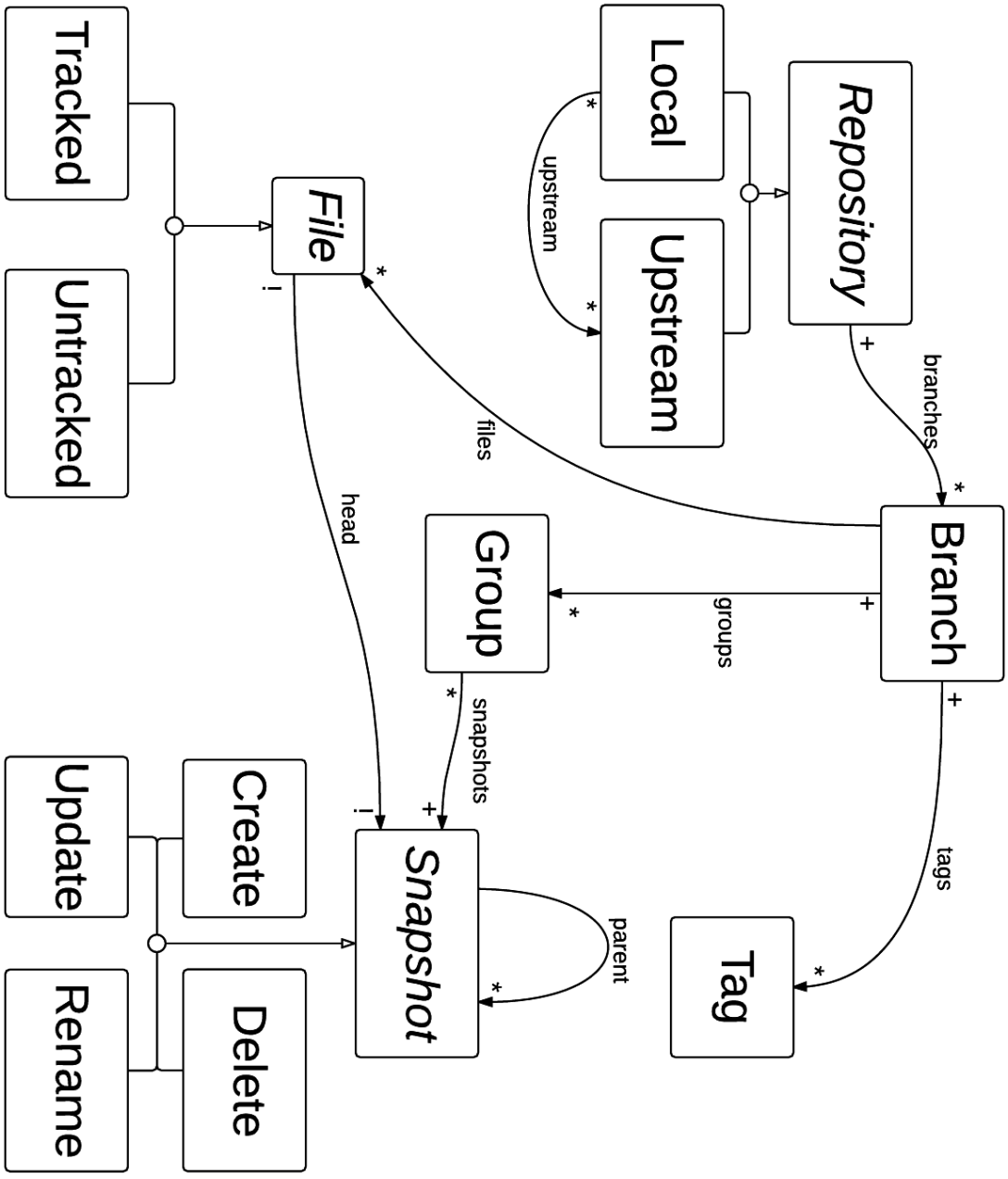

Figure 3-1: Concept Model of Snapstore.

distinction dictates the values of the snapshot fields. Create snapshots have no parent. Update snapshots have a parent, a child, and content. Rename snapshots have a different filename than their parent. Delete snapshots have no content.

All snapshots have parent snapshots and child snapshots. The snapshots of a file are related by the tree they create with these parent/child relationships. This ordering forms the snapshot tree described in the tutorial section. The snapshot tree is guaranteed to be an in-order description of snapshots a specific client has made to a file in a branch. Each unique (branch, file) tuple is represented by its own snapshot tree. A snapshot can have multiple parents if it is the result of a merge operation.

Any file, identified by its snapshot tree, can either be tracked or untracked. Untracked files will not create snapshots and will not affect the local or upstream repositories.

3.2 Grouping Changes - Group

A *group* is an assembly of related snapshots. A group must contain at least one snapshot, but there are no restrictions on what kinds of snapshots can be in the group or what their relationship must be. The same snapshot can exist in more than one group. It is up to the user to decide what makes a group of snapshots logically related. This allows flexibility in projects and development strategy.

Groups are an attribute of a specific branch. Even if a group contains the same snapshots across different branches, they are treated as different concepts because they exist on independent lines of development. They can be given names for identification for the user.

3.3 Recording Coherent Points - Tag

The notion of a *tag* allows users to label logical milestones in their work. They describe groups but have an added function over a group's name: they describe the status of the branch as coherent. Here, coherent means that the project is state that

is ready for further development or work, though the definition will differ from team to team [?].

Tags will always describe groups that are perfectly verticle. That is, one snapshot from every snapshot tree in the branch is used. An example of this is tagging every head snapshot in a branch at a given time with the tag "Submitted to Scientific Journal".

Tags are also an attribute of the branch. This means that they must be created inside a of an independent line of development. They can be copied across branches when merging and cloning, but they stay a fundamental attribute of the branch concept.

3.4 Support Parallel Lines - Branch

In Snapstore, the concept of a *branch* supports parallel and independent lines of development. These branches are completely separate from each other. The facilitate the appropriate partitioning of data.

The branch houses three of the other main concepts in Snapstore: snapshots, groups, and tags. All three of those concepts exist within the confines of a branch. All of these things together consitute a line of development, and so the branch is the conceptual representation of that line.

Branches make up a repository, whether that repository is local or upstream. Switching between them constitutes changing the line of development, project, folder, or anything that delineates the user's work. Switching between two branches on a single local repository that are stored on different upstreams has no averse effects due to their independence.

Branches can be merged together, synchronizing the parallel development. This simply involves combining each branches individual snapshot, group, and tag data together.

3.5 Synchronize Changes of Collaborators - Upstream Repository

Snapstore uses a centralized data storage system that holds all connected branch data, called an *upstream repository*, or upstream for short. While users do not necessarily have to use an upstream for a given branch, it is the only way to collaborate with other users on that branch.

Every branch can have multiple upstreams. This allows the user to distribute where their snapshots, groups, and tags are saved and backed up. These upstreams can then be shared with other Snapstore users, provided that the upstream is connected to those users.

All changes that occur at the branch level (branches, snapshots, groups, tags) are reflected in any connected upstream. There, the upstream can see if any other users have access to the branch and it can push the changes down to them.

3.6 Disconnected Work - Local Repository

The ability to leverage the benefits of a version control system without needing an internet or network connection is handled by the *local repository*. The local repository affords all of the same relationships to other concepts as the upstream. That is, it is a collection of branches, which are in turn collections of snapshots, groups, and tags.

The local repository has one other important relationship, it can have zero or more upstreams attached to it. These upstreams are mirroring the data housed by the local repository.

3.6.1 Discussion

During the design process, there were many decisions made that had lasting tradeoffs for Snapstore. Three of those tradeoffs are explored below.

Granularity of a Snapshot

The decision of what a snapshot would represent was one of the first design decisions that we encountered. In many VCS systems such as Git and Mercurial, saving changes involves creating a new commit that includes the state of every file in the branch. We could not create Snapshots that represented every file in a branch because we realized the necessity of the group concept. If a snapshot described every file in a branch, this snapshot would also be a group, resulting in an overloaded concept [**?**]. We separated the idea of saving changes with grouping those changes. This way, users can group together changes to a file over time.

Another reason the snapshot describes only a single file was that it was more intuitive to a typical user. If a user was to save a file and create a snapshot, they would expect that snapshot to relate to the thing they just interacted with, that file. They would not expect it to relate to every file in the branch.

One downside of this approach is additional computation when finding the current state of a branch. In Git, the current state is the head commit. In Snapstore, we need to calculate this by grabbing all of the head snapshots for a branch. This is typically trivial, so the tradeoff is beneficial.

The Upstream

Different version control systems and file sharing systems have their own ways of handling storage. Git, for example, use a decentralized storage system. Dropbox and SVN, on the other hand, use a centralized system. When deciding which model to use for our upstream, we looked at both models' pros and cons.

Snapstore uses a hybrid centralized/decentralized upstream model. On the surface, it seems centralzied because all collaboration must take place via the upstream repository. We wanted this centralized feeling in Snapstore because the centralized model is easier for the majority of users to use, and it has a faster learning curve [**?**].

However, Snapstore users have a local repository, where actions can be made without requiring an active connection to the upstream. Snapstore users each have

an entire history of their branch on their own system, just like in a decentralized version control system. Because of this, users can work offline, without checking out a central repository.

There are downsides to this hybrid model. Initially, there is only one upstream server, and so the location of the data is held by a single entity. If users want to set up their own server, they must use a machine to do so, and there is the overhead of setting up that server. Because there is no push/pull model in Snapstore, this machine must always be online in order to facilitate collaboration. Further, users cannot share directly with subsets of users on a given branch. They can, however, work around this limitation by creating new branches with new sets of users.

We used our goal of opt-in complexity to create a balance between the simplicity of a client/server, centralized model with some of the power of a decentralized system. In Snapstore, the data on the upstream is indeed "blessed", and so the model seems centralized. However, by offering a local repository alongisde and potentially independent from the upstream, Snapstore has some characteristics of a decentralized system.

File Names

Whether or not to make the file name a property of a file or the identifier for a file was an important decision for Snapstore. Systems like Git use the file name as an identifier for a file. Because of this decision, renames to a file are sometimes processed as deleting that file and creating a new file.

We needed to be able to fully support renaming files in Snapstore, so file names in Snapstore are simply a property of the file. Each snapshot in a given snapshot tree will have the same file id. When branch merging occurs, only snapshot trees with the same file id will be merged. The merge operation will look back over a file's two snapshot trees for a snapshot with the same id. That snapshot is the common ancestor. This allows Snapstore to accurately handle renames and merges.

Creating Branches

The way we create branches also changed during the design of Snapstore. Initially, a branch was always created with empty snapshot, group, and tag collections. This enabled shared branches and individualized version control, but it wasn't enough to support merging. Because file names are properties of the file, snapshot and file ids needed to be brought over when creating a branch in order to facilitate finding the common ancestor when merging.

To fix these issues, the Snapstore branch became similar to Github's fork. Creating a branch always involves forking from another branch (a brand new branch forks an empty branch). This allows a subset all of the snapshots from the original branch to be brought into the new branch so a common ancestor can be found in the event of a merge.