Idee (revised)

* Verwendung verschiedener domainKeys (Siehe auch Ppt)
* **Job Procedure Domains**
  + Jeder Job hat eine Menge von Procedures, welche nacheinander ausgeführt werden. Jede Procedure hat eine Domain der Form
    - **Job\_<X>\_Procedure\_<Name>\_<N>**
    - <X> ist eine eindeutige ID bestehend aus Job + currentTimeInMs() + random.nextLong()
    - <Name> ist der procedure.getClass().getSimpleName() der Procedure
    - <N> ist der currentProcedureIndex() im Job, d.h. der Index in der ArrayList<Procedure> der im Moment zu bearbeitenden Procedure.
  + Jeder Executor hat einen eigenen Domain für jeden Task (**ExecutorTaskDomain**), in welchem die Taskresultate ({<K, {V}>}) abgespeichert werden
    - Wenn ein Executor mehrmals denselben Task ausführt, kreiert er auch mehrere Domains
      * **ExecutorTaskDomain**: **„Executor\_<PeerId>\_<StatusIndex>\_Job\_<X>\_Procedure\_<Name>\_<N>“**
        + PeerId ist gegeben dadurch, dass jeder Executor mit 1 Peer verbunden ist
        + StatusIndex bezieht sich auf die „EXECUTING\_TASK“/ “FINISHED\_TASK“ Position innerhalb einer Multimap<PeerAddress, STATUS> in jeder Task. D.h. wenn der Executor einen task mehrmals ausführt, wird ein weiterer Status zur Multimap hinzugefügt. Die Position in der Values Kollektion dieser Multimap ist der StatusIndex
  + **Spezialfall** Job Submitter:
    - benimmt sich wie ein Job Executor für die
      * **Verteilung der Daten**: Key = Filename, Values = {File}

Foreach(FileName){

DHT.add(**Filename**, **{File}**).domainKey(**Submitter\_<PeerId>\_0\_Job\_<X>\_Procedure\_<Name>\_<N>**);

}

* + - * **Verteilung der Keys**:

DHT.add(KEYS, {FileName}).domainKey(**Job\_<X>\_Procedure\_<Name>\_<N>)**

* + - * **Verteilung der ExecutorTaskDomains** (Datenanbieter):

DHT.add(FileName, Submitter\_<PeerId>\_0).domainKey(**Job\_<X>\_Procedure\_<Name>\_<N>)**

* + - Nach Datenverteilung: JobSubmitter versendet “New Job” Broadcast mit Job bestehend aus <[job-ID=Job\_X], List<Procedure>, CurrentProcedureIndex>
      * Alle Job Executor, die den Job noch nicht erhalten haben, tun diesen auf ihre Job Queue.
      * Alle Job Executor führen Job für sich aus
        + Task Scheduling wichtig (siehe unten)
      * Nächster Job in Queue wird genommen und gestartet
        + Jobs werden der Reihe nach gestartet, d.h. frühere Jobs werden vor späteren Jobs gestartet.
      * Task Scheduling: jeder Job Executor definiert seine eigenen Tasks (am einfachsten: task-id = KEY):
        + Zu Beginn sind keine Tasks definiert für den Job
        + Nach und nach werden für die eingegangenen Keys Tasks definiert (zufällige Auswahl, solange noch nicht alle tasks definiert sind)
        + Durch Synchronisierung mit anderen Job Executors kommen ggf. nach und nach neue Tasks rein (da alle Job Executors dieselben Tasks generieren werden): Scheduling wird nicht-definierte Tasks diesen Tasks bevorzugen
        + Je nachdem, wie viele JobExecutor das gleiche Resultat für einen Task erhalten müssen (definiert in Job, via Hash) werden nach und nach weniger Tasks geschedult, bis alle Tasks als gefinished markiert sind
        + Wenn dies der Fall ist: FINISHED\_ALL\_TASKS gebroadcastet
      * Job Executor holt sich Keys für Job mit DHT.get (siehe PPT)

**jobProcedureDomain = job.id()+”\_”job.nextProcedure(job.currentProcedureIndex()**)+“\_“+**job.currentProcedureIndex()**

{key} = DHT.get(**KEYS**).domainKey(jobProcedureDomain)

{executorTaskDomain} = DHT.get({key}).domanKey(jobProcedureDomain)

{values} = DHT.get({key}).domanKey({executorTaskDomain})

* + - * + **Alternativen (Datenlokalität)**: statt von Beginn an über DHT.get() versucht JobExecutor erst über **StorageLayer** Daten für den Job\_Procedure abzufen, die auf seiner Disk gespeichert sind (Sowohl KEYS, als auch Providers für diese KEYS, und dann auch die Daten für diese KEYS von den Providers (bevorzugt sich selbst gegenüber anderen)

Wenn nicht alle Daten verfügbar für einen Key auf dieser Disk: restliche Daten via DHT.get() anfordern. Trotzdem diese Daten bevorzugen, da mehr lokal gespeichert sind

Nur wenn alle lokalen Daten für diesen Job\_Procedure aufgebraucht sind und Job\_Procedure noch nicht beendet ist: DHT.get() verwenden

* + - * Job Executor führt die Tasks aus, bis Scheduling keine Tasks mehr ergibt (dann würde dieser Job Executor ein FINISHED\_ALL\_TASKS broadcasten, was alle anderen Job Executors dazu veranlasst, mit der Ausführung aufzuhören. Der Job wird dann mit dem Job, der mittels FINISHED\_ALL\_TASKS gebroadcastet wurde, synchronisiert (Das heisst, alle Tasks geben dann an, welche ExecutorTaskDomain für die nächste Prozedur gewählt wird (für jeden Task jeweils nur einen!). Zusätzlich bieten die Tasks alle abgelehnten ExecutorTaskDomains zum Aufräumen an (um Speicherplatz freizugeben). Erst jetzt werden die KEYS & zugehörige ExecutorTaskDomains in die jeweiligen nächsten JobProcedure domains gelegt
      * Task Ausführung bedeutet, dass neue {<K,{V}>} generiert werden von den Inputdaten. Diese werden in der nächsten Domain (Procedure<N+1>) gespeichert, das heisst:
        + jobProcedureDomain2 = **Job\_<X>\_Procedure\_<Name+1>\_<N+1>**
        + executorTaskDomain = **„Executor\_<PeerId>\_<StatusIndex>\_Job\_<X>\_Procedure\_<Name+1>\_<N+1>“**
      * Sobald Finished all tasks & Synchronisiert:
        + Entweder currentProcedureIndex ist inkrementiert und alles beginnt von vorne, mit neuen Daten und neuer Prozedur
        + Oder JobSubmitter ist via Broadcast informiert, wo die finalen Daten der letzen Procedure zu finden sind (Beispiel: Job1, Wordcount: 2 Prozeduren, am Ende wäre alle am Ort Job\_1\_null\_2 gespeichert (null, weil die letzte Prozedur immer leer ist, das heisst nichts mehr ausgeführt werden kann. Kann man dann auch anders nennen)

Nun kann JobSubmitter genau gleich die KEYS abfragen mittels Domain (Job\_1\_null\_2), dann die Provider, und dann die eigentlichen Daten.

* + - **Maintenance**: So ein vorgehen produziert viel Abfall (alle executorTaskDomains, die fehlschlagen, nicht fertig werden, etc. haben ggf. unvollständige Daten produziert) 🡪 Bei FINISHED\_ALL\_TASKS wird ja der finale JobProcedure gebroadcastet und alle Peers wissen dann, welche ExecutorTaskDomain für die nächste JobProcedure verwendet werden. Beispielsweise kann man mit jedem JobExecutor auch noch einen „Cleanup“ starten (eigener Prozess, nicht nur Thread), welcher ebenfalls an der DHT hängt und jeweils aus den Tasks die abgewiesenen Domains herausnimmt und die Daten dafür aus der DHT mit remove entfernt, unabhängig von den JobExecutors (Wenn man die auf jedem Peer Computer laufen lässt, hat man auch Garantie, dass alles läuft wenn wenigstens 1 Peer übrig ist)