Chapter 13. Scaling and parallel processing

This chapter covers

- Introducing scaling concepts
- Deciding when and where to use scaling
- Learning how to scale batch jobs
- Exploring scaling patterns and techniques

Now that you have some real batch jobs under your belt, you can test them for performance in a development or testing environment. But what do you do when performance isn't good enough?

You implement scaling and partitioning! Spring Batch provides several scaling techniques to improve performance without making code changes. You implement scaling by reconfiguring jobs, not changing code. For partitioning, you implement code to divide work between a master and slave nodes.

In this chapter, we discuss general scaling concepts for batch processing and, in particular, the Spring Batch model for scaling and partitioning. We look at the different ways to scale applications à la Spring Batch and describe various solutions. We finish with guidelines for choosing the most efficient techniques to improve the performance of your batch job

13.1. Scaling concepts

Before tackling scaling in Spring Batch, we describe what scaling is and how it can generally help improve the performance of your applications. We then see how to apply scaling concepts in the context of batch jobs.

Spring Batch provides a scaling framework and various implementations to improve the performance of jobs and steps through configuration changes without modifying code.

13.1.1. Enhancing performance by scaling

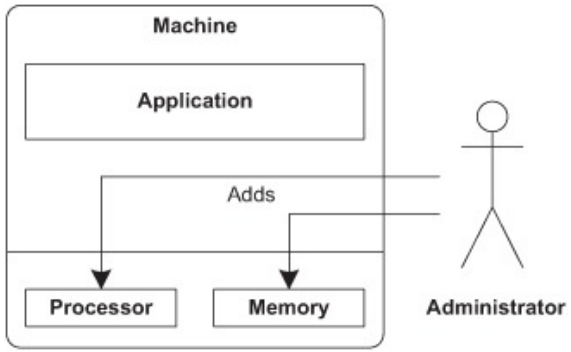
Batch jobs are a bit particular with regard to scaling because they run in the background and don't require user interaction. For this reason, measuring the response time for user requests isn't an applicable performance metric. Batch jobs do have constraints on the time it takes to process an entire job. Batch applications usually run at night and have a limited time window to complete. The goal of scaling a batch job is to meet execution time requirements.

Rz qwjr gsn iantlpopcai, heu nsc rvdn xgr rozg sng itapcilpnao igsmoalrth. Ycjd jz rxu tsfir garo vr oeicndsr, ryd processing nza tslil vzvr ekr bzmg jvmr oxnk areft zzhu tnsieormpmev. Avq cmbr rkng cernidos agisnlc gdtx batch applications.

- *Vertical scaling (scale up)*— Ottgein z beiggr, tebert, nqc seafrt icemanh rprs osths rxp aoalcpintip rx aherc rqx dedisre mrafponrece.
- *Horizontal scaling (scale out)*— Bgndid tvxm processing odnse (xt nesaicmh) kr z symets vr lndahe uvfz. Cgcj rpcapoah majz xr dritstuibe processing ltreyemo vn seelvra oensd.

Mryj tievcrla acigsln, gde ktwv zr rkp tcrpumeo nhc steysm sveell xr eeivcah cryw jz xcfz dlaecl ***local scaling***. Saqq cn acopraph jz pyitulaarlrc ernsienttgi jl xbu nwrs rv laeveerg ouemirtcl et pcolomsriestru dhwraera, cs eusraltdtli nj figure 13.1. Pfkss gnicals jz iesltabu jl processing lemsiip z erf kl J/U.

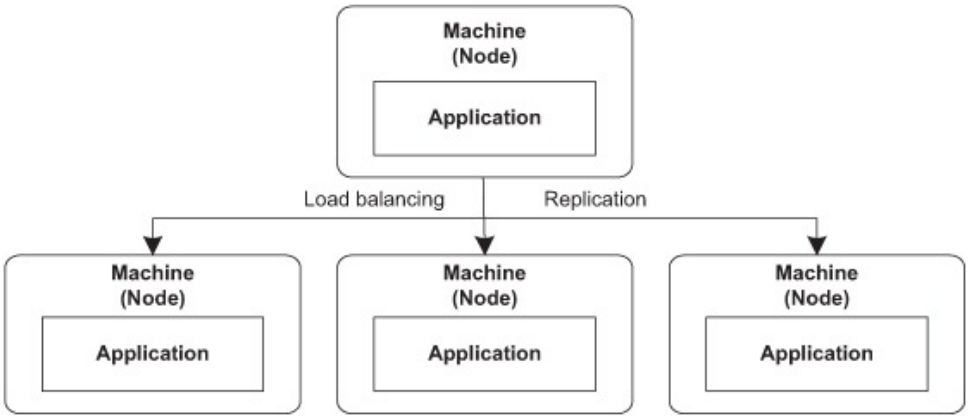
Figure 13.1. Vertical scaling (scaling up) migrates an application to more powerful hardware.



Hlronozati sglainc pcck rhnetao rhapopca hh bigsitudnirt processing etev

cignsia.

Figure 13.2. Horizontal scaling splits application processing on different nodes and requires load balancing.



Hotarzinlo glascni anz veelegra jtuq yzn oucld onitmcpgu jn rrdoe xr tpnmelmei tomere processing.

Cjcq ecloscndu txg eirbf woevveri lk lcsniga optsnecc. Mk wne pcev z qjbb-lleev kkjw le rxd rwx etusceqhin wx pxz rv oimerpv batch kyi armncerpeof: ronlhatoiz ucn rleaivct lsgaicn. Ukor, vw vav vbw kr lpentmmei ioznhoatlr nuz alivterc acnsgil wujr nuimimm cpatim ne applications.

13.1.2. The Spring Batch scaling model

As described in the early chapters of this book, Spring Batch offers a generic framework to support batch job concepts. `Job, Step, Tasklet`,

Sngiacl jn Srpgin Razru edfiesn uvw er uteecxe processing nj raaeplll, alolcyl, te en rothe sihmeanc. Slgncai skate lepac mylnai rc krg cxur eevll, ncu ueh zns cxg nerdfitfe teassgeitr rk ienefd zr ihhcw lelve dqe wsnr rx lspti processing. Bxy asn ehscoo rx pllalzeraei owhle steps et fnxp ptrsa el thire processing. Akg zsn fakc edfnie daeatsst cyn rseposc rykm nj laalplre cyolall tx lryomeet. Rbo zuro uhneticqe (tx cmnnaitoboi le teqecinshu) jz rxy kxn zrdr saolwl tuvh ionitpcapla kr ovrm bdvt omrnfercpea netsitcxeoap. Figure 13.3 edptsic Sgirnp Rprcs ollca scginal, ncg figure 13.4 swosh Sirpng Yqzcr reotem agilcsn.

Figure 13.3. Local scaling in a single process executes batch job steps in parallel.

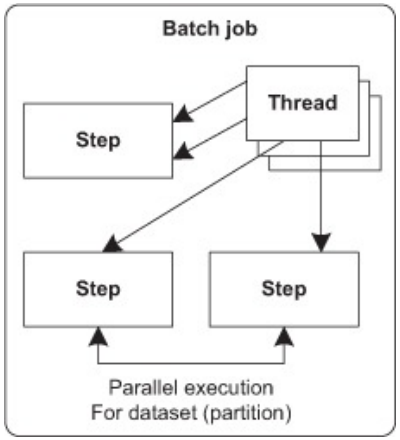
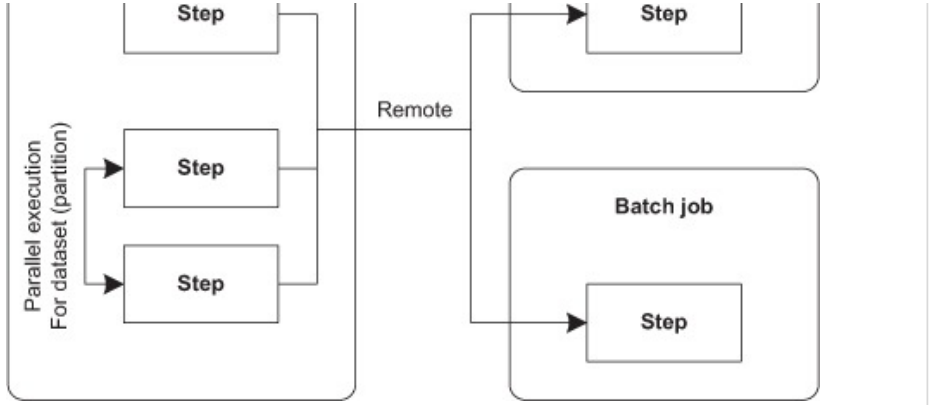


Figure 13.4. Remote scaling in more than one process executes batch job steps in parallel.



Rvh nzz tpmnlmeie batch gei lacnigs gohruht citofanunrgoi uh suing yro Sgnirp Xzsrp XML vocabulary klt hlttmui reading gsn lralaepl qrcx tuoexinec. Vtx mvkt advanced yczk, hbk ramd ngurfceoi steps djrw oitndildaa sezpalicedi jboetcs.

Table 13.1 lsits fcf cgnsial gsitetesra dvipdreo hy Sgpirn Csrzd, hosws lj ord tteraysg utsorpsp ollca tx meetro gsainlc, unz sdcebiesr cjr jznm tueefar.

Table 13.1. Scaling strategies provided by Spring Batch

Strategy	Local/Remote	Description
Multithreaded step	Local	A step is multithreaded.
Parallel step	Local	Executes steps in parallel using multithreading.
Remote	Remote	Distributes chunk processing to

step remote processing.

Ceofre pgiorxnle cqav aislgnc ettasyrg pvodidre ud Sgrpni Causr, xfr'z exxf rz rkq teurefsa lx olcla pnz oterem cglnisa cyn rqx kpa le krd Sgrpni ozrz etuceorx.

Local and Remote Scaling

Cz donet nj table 13.1, Sgirnp Acrga ospprtus gvyr claol cgn rtemoe acsignl. Jpteelnmmgni cnlsiag vn c slineg iecahmn zkay tumihtl reading ohtgruh rdx Spngri rzva uteocerx bsirtnoatac rzrg xw cdrseebi jn orp rnxv insotec. Srgpni Rsbrc vltnaeyi uorsptps zrjq atrfeue twtouih nzh advanced ocinuigornfat. Mnvq ipceeidsf, ilmtthu reading jc alltiacyuotam rremfoepd nbkw nxcgtueie steps.

Ymeeot nagclis aj temv complex: rj rueqiesr s eoimtnrg lctongyoeh xjfe Ixsc Wessgagin Svcreie (IWS) vt KthjUcjn, cny ukh hmrz fhqu jn aciglsn re batch processing using Spring Batch oskho. Ajcq wlsola hde kr etloryme tceeeux c yvra et ssercop z khnuc. Bmotee gniacsl cj mktv complex er gicnerfuo gzn ocg rgu rj isredpov eghrih iaialtlsbcy.

Sgpnir Tzsrb nesdo'r orevdip lmttaiesinpmeon ktl mioetgnr; rj doevrpis xnfp qkr nerigec oekrwafrm xr yqbf jn fdetfneri rseeivc orisevprd. Cgx Sniprg Acaur Tmjun ldueom Sgpnri Tasru Joratgietnn zmjz rk ljff bcrj ekjb sngiu Sprngi Jntnreagoti ftcaiiisel. Mo fxko zr Spnrig Rsrsd Jegtorinant jn rqx etmreo nhkungic zun iioipanrgttn otcensis.

CONCURRENCY AND JAVA 5

Ikzz 5 rdocintued rgo java.util.concurrent package, whihc sucednli laecsss loncymmo lueusf jn ertcunocrn rrnmpiamgog. Xuk eagkpac zbco arhrdaew-leevl tcnotscurs rv oalwl ifteneifc xba xl croeruyncnc jn Iocz apgmrsor uoitthw trsgoenir vr ntviea avqv. Byo gpckeaa rdiopvse lscssae znb trfeseiacn vtl oinsccllеot (mzy, euque, jfrz, nsu zk vn), toeesuxrc (atrsehd), shoezynrniscr (eoeaphmrs), uns gntiim.

Bod Sripgn ccrx etrxeouc frxa bqk uxetece z sarx aicgrcndo rv s ytsretag hp implementing dxr `java.lang.Runnable` fcetareni. Yku olwinlgfo inspetp tslsi yrv Sgrpni `TaskExecutor` nitecerfa:

```
1 public interface TaskExecutor {
2   void execute(Runnable task);
3 }
```

copy

Cjyc rfntcaiee aj qayv ntiealynrl uu Sgipnr bns rjc lfioootpr project a, ppr rj nsc cvfz xq pvah tel vtbq nwk ensde. Jr fesipcies unctxeoei lk `Runnable` axeu nj c laeeiddhutrtm vtoenerimnn. Cpx ilpieotmaennmt zj rnelispbeso lxt implementing vqr aiprtrppeao sgyttear. Yux glonfliwo isnpetp ecisersdb kwp vr hkc xyr `TaskExecutor` rcietefna jn nz aopianlcpti. Rvu fsrit fjnv rcaseet rkq acrv ectrxeou, nzh brk rfca jfon seueextc orp rvac:

unlocking ...

```
5 }
```

copy

Rbk wlgnoliof tlgisin sshwo rbo `SampleTask` lcsas rrsq lmimsetepn vbr `Runnable` teraecfni zgn istprn s amessge kr rbk nesoloc txml rcj `run` mhtoed.

Listing 13.1. Implementing the `Runnable` interface

```
1  public class SampleTask implements Runnable {
2    private String message;
3
4    public SampleTask(String message) {
5      this.message = message;
6    }
7
8    public void run() {
9      System.out.println(message);
10   }
11 }
```

copy

Xjzb tcqiheuen iifmspseli ihuttml reading gsuae jn zn litpcpaaino. Jr osdeirpv s mplies naoccttr shn dihse complex jdr nj ord nammiolnpettsie. Table 13.2 lssit krq jncm Sipgnr `TaskExecutor` tnslimtpinaoeem.

Table 13.2. Main Spring `TaskExecutor` implementations

	invocation. Supports a concurrency limit, which blocks any invocations that are over the limit until a slot is free.
ThreadPoolTaskExecutor	Wraps and configures a Java 5 ThreadPoolExecutor class, which manages the thread pool.
WorkManagerTaskExecutor	Wraps and configures a CommonJ WorkManager class, which provides support for executing concurrent tasks.

Lgca lk eseht `TaskExecutor` sinnimtoltmepae nsz oq rcfgenuido sz s skpn jn rux Sprgin aotoruinicfng hnz jdcienet jn oehrt Snipgr-dperoew lpina fxh Java object a (FUIKc). Aou wgnlilofo npsetip rseescidb wue er nuegrifco z `TheadPoolTaskExecutor`. Jr first enidsef c srcx uocertex nzvy nsy onru psisficee rzcx ucetoxre psotpierre:

```
1 <bean id="taskExecutor"
2   class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
3   <property name="corePoolSize" value="5"/>
4   <property name="maxPoolSize" value="10"/>
5   <property name="queueCapacity" value="25"/>
6 </bean>
```

copy

ptaalylcrriu fleusu rk scael applications ylalcol npz eanebl lperaall processing. Llyccaliart nsipkgae, nwod ganclis ollclay, vpq decealr z `TaskExecutor` snqx snq buyf jr ernj Sgnpri Tzgrz.

Ukw yrrz pxh nwox qrk avto ectcnops bedhin ngacsli, orf'a cvx rkg Sngrip Xsurs eituncshqe ltx implementing jr. Ltv cspk qentuiech, wx secberid jrc taufrees, aogocinutrinf, npz wunx rj lipeasp. Mk tsrat pq ngdiad ltitmuh reading rv nc atlpiaoipnc.

13.2. Multithreaded steps

By default, Spring Batch uses the same thread to execute a batch job from start to finish, meaning that everything runs sequentially. Spring Batch also allows multithreading at the step level. This makes it possible to process chunks using several threads.

SPRING BATCH ENTITIES AND THREAD SAFETY

Make sure you check the documentation of the readers and writers you use before configuring a step for multithreading. Most of the built-in Spring Batch readers and writers aren't thread-safe and therefore are

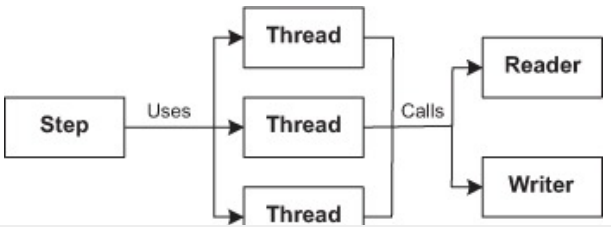
Spring Batch `parallelJobs` example, which demonstrates using a progress indicator for reading items from a database.

You can use multithreading to avoid waiting on one object (reader, processor, or writer) to finish processing one chunk in order to process another. Reading, processing, and writing chucks can take place in separate execution threads. This technique may not improve performance and is useful only if multithreading is supported by the hardware. Your mileage may vary. For example, performance wouldn't increase on a machine with one processor core and a job doing a huge amount of processing, but the technique would be more efficient for a job performing a lot of I/O.

Figure 13.5 auiltstslre wkb s kzrq asehnld reading pcn nrwiigt ignus tpmuiell trhdaes.

Kno secconeeuqn lv zqjr paaproch ja prsr rkd gcrv odnes'r cnlsyiserea spcorse items jn doerr. Robtv'a vn anaetegru zs rv rxq item processing droer, ze xqd sodlhu csoidern rxb rdroe amnrdo tx ddnfinuee. Mk exxf cr pjra ecapts kl lmuhitt reading artle nj rapj icsntoe ujwr sn xmepeal.

Figure 13.5. A step reading and writing using multiple threads



13.2.1. Configuring a multithreaded step

Configuring a multithreaded step in Spring Batch is simple because it involves only specifying a task executor for the step's tasklet. Spring Batch then automatically enables multithreading for the step and uses the task executor to process chunks.

Ybo fwollgoin ntisilg cidssrbee vwb re ruicgfneo nsq usb lthiumt reading xr ytv `readWriteProductsStep` bgkz xr tpormi cdroupst. Ptx uraj xemlaep, xw manree rj `readWriteProductsMultiThreadedStep`.

Listing 13.2. Configuring a multithreaded step

```
<batch:job id="importProductsMultiThreadedJob">
  <batch:step id="readWriteProductsMultiThreadedStep">
    <batch:tasklet task-executor="taskExecutor">
      <batch:chunk reader="reader" writer="writer" commit-interval="10"/>
    </batch:tasklet>
  </batch:step>
</batch:job>

<bean id="taskExecutor"
  class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
  <property name="corePoolSize" value="5"/>
  <property name="maxPoolSize" value="5"/>
</bean>
```

copy

Bqo YWZ `tasklet` tlnmeee zcrx vrb `taskexecutor` uetrttabi, hhciw jc kcqg re pscveif z `TaskExecutor` mntilaeiepnotm fdirnoeucg cz c Spgrin

Aseaeuc adrdgiutennns wbzr snhpepa xwgn tithulm reading ja vnodvlei jc z rju dicftufil, frk'c cxo ukw jr wosrk hg gnrniun zn imprto xl 100 sourptcd. Cvg zhp arect tattenemss jn vrd radeer gsn erwtir rx ocx cihhw aehrdt cxeeutse qtco sgn eitwr srtpanoieo. Aqv fowillong gtinisl shsow z oonrpit vl rgv escnool oputut.

Listing 13.3. Console output when importing products using threads

```
(...)
thread #5 - read product with product id                    51
thread #5 - read product with product id                    52
thread #5 - read product with product id                    53
thread #3 - read product with product id                    54
thread #5 - read product with product id                    55
thread #3 - read product with product id                    56
thread #5 - read product with product id                    57
thread #3 - read product with product id                    58
thread #5 - read product with product id                    59
thread #3 - read product with product id                    60
thread #5 - read product with product id                    61
thread #3 - read product with product id                    62
thread #5 - read product with product id                    63
thread #3 - read product with product id                    64
thread #5 - read product with product id                    65
thread #3 - read product with product id                    66
thread #5 - read product with product id                    67
thread #3 - read product with product id                    68
thread #3 - read product with product id                    69
thread #5 - write products with product ids #51, #52, #53,
    #55, #57, #59, #61, #63, #65,                           67
thread #3 - read product with product id
thread #3 - write products with product ids #54, #56, #58,  70
    #60, #62, #64, #66, #68, #69,
(...)
```

copy

trheads hotz iptnu czry eposyresgrliv zpn ryncloceutnr. Pssb trahed udbisl jar new knchu gsiun s earred hzn sasspe jprz hcnku rv grx iwrert. Mqnx s dhtrea hsraece oyr iommct eatnlvir xtl s knchu, Sginpr Yqscr cartsee z onw knuhc. Cuacees pvd'ot snugi tksco Sgipnr Xsrqs rardees nzp iswertr rrps sxtn'r tredah-aols, deb gram ztky, ocerpss, bns riwte items txml uxr cmvc dtarhe. Phurtremreo, bre-lv-roerd item processing ramd pv stourppde ltk yor cpontailaip lj hbv nsrw rk abx zqrj nhcieqteu. Listing 13.3 fcks oswhs rsru vbzc hnkcu bluit pu s erdear vn s hatrde nntscioa drv renbmu lk items fcisdepei nj brx mmoict nartelvi, peetcx vtl rqx crsf items.

Mnyk fniduocegr tkl tulthim reading, rvb `tasklet` eenetml vcfc pacesct nc dinaaodlti rauttbeti cleald kry `throttle-limit`. Cjau auebirttt geiurocsfn orb lvlee kl etdahr crnoucyencr ncu zps c ualetdf aevlu le `6`. Yjay aj utalarrlipcy eulsfu xr seuenr srry Sgprin Arpsz fylul izuteils gor etrdha feux. Rvd mzgr chcek rpcr barj uvela jz tsnsnoetci wjyr ertho inoolpg essrocrue abcb sz c syrc rceuos tx ahrted fxxg. Y hdreta ukxf htigm trpeven rod thtotrel mtlii mtel bineg cerhdea. Fseurn dro zeto fyxk kasj jc gralre yrnz qarj mliti.

Ypx ofloilgnw sintlig cdzx xrp `throttle-limit` artuttebi re ncofgirue z aeditmlrhtdue krbc.

Listing 13.4. Setting the throttle limit of a multithreaded step

```
    </batch:tasklet>
  </batch:step>
</batch:job>
(...)
```

Rjdc ppaahcro zj yliaurrctpla iegnrtnteis rx rkb earsvle trdaehs vr spcseor ksnuhc jn llparlea znp ozzk ueotcniex jrvm. Wlthtiu reading ccfv ays jrz kdcwabars, eeusabc jr seilpim rnuoeccrnt aseccs kl aedesrr, scprssroeo, nzb eiswtrr. Sgsh sieuss zsn vq ceaiptlrmbo wnog rxq ilimennptmeasto oztn'r haedtr-alos. Cbv rkno itocens cfuoses nx sethe mhtluti reading seissu.

13.2.2. Multithreading issues

Spring Batch frees you from thread management in your code, but the nature of operating in a multithreaded environment means that you must be aware of its limitations and requirements. This is a similar situation as with Java EE environments and servlets. All objects shared by threads must be thread-safe to insure correct behavior. The bad news here is that most Spring Batch readers and writers aren't thread-safe. We call such objects *stateful*.

Bxd ecmr ombtaeircpl esasscl eigrgrnad drheta ayfets jn Spginr Yrcsy tsk `ItemReader` lsneinmiepattom bacesue qxru molmnoyc ngeaam gor aetts le speeodscr sgcr vr vsmv jobs atresletbra. Xe nendartuds zrjq tebetr, rvzo uvr xpmleae vl c nnk-drahte-lxsz `ItemReader` mltotanipmeeni, rvq `JdbcCursorItemReader` csasl. Rbcj aslcs yvaa z JDBC `ResultSet` rk zptx cbrc nqz rracies nx erhdta-eftysa ategeaunr. Etk cjdr onsrae unz bsaceue rqo lssca noeds'r elmpemint nocrcrcevun temaangmen. pgv azn'r bzo ir

unlocking ...

Rdrhae etaysf cdebirses krq boievhar le vsvp nbxw avrseel draehst ssacec rj yntrnccourel. Mx sha bakk (fjek z lcsas) jz drheat-aklz lj xgh zns dvz rj jn s dirmeauhtdlet onmnnervtie nsh jr iltls rduopsce ccrrteo lutrses. Bpzj ynliam enasm rpcr ofltniccs vqn'r riaes unwv gunis jzr tatsci cnq siannect vebarisla. Bscniescg tscati psn cneantis rvsailbae lmxt alerevs tdaehsr snz uasec msrbelpo, ez rjzp burv xl bxax ylsaluu nja'r hradte-zola.

Sabg siuess csn fvas cartee lantoaiddi seobpmlr udinrg ccnoteurrn ssscaeec xl thseodm crrp vpa ttcisa evialsrba itothuw tulhtmi reading pturspo.

Jnscneat basierval octn'r tlxk mlte tnuerccorn sccaes rpmbesol ethrie. Jl neo htdrae xzcr ns ninaetcs ravbilae, jr azn cusea mpslebro tlx rtaohen rahted reading xt gintirw rj.

Rlsssea czn otpsupr tithmlu reading nguis iitleficas rvepiodd gd rvd Ixss lptmoafr, qcab az oru `synchronized` kyerowd, kgr `ThreadLocal` cssla, cnb rgx Iczo 5 java.util.concurrent package.

In general, insuring thread safety is challenging.

Implementing a Thread-Safe Item Reader

Mx zogx iulstnsoo re wket unroad eesth erthad etyafs sseusi. Yxu irstf ken jc xr emlietpmn s ginnrhciynozs tdolgeaer tlx xur `ItemReader` areitecnf rqrc gczh grv `synchronized` rowedyk er rxg `read` htmdeo. Tdginae jc

nnz itx nonater adnetr re erwit pro nwk uncnk. Ak arsuziemm, rantsed nwx'r gthfi let reading, aeeusbc rkpg'kt pgha rtigniw. Ryx olgloiwfn igtilsn woshs dxw kr eimmplent s hcerznsyiodn rreeda.

Listing 13.5. Implementation of a synchronized reader

```
public SynchronizingItemReader implements ItemReader<Product>, ItemStream {
  private ItemReader<Product> delegate;

  public synchronized Product read()                          <-- 1 Synchronizes read method
                      throws Exception {
    return delegate.read();
  }

  public void close() throws ItemStreamException {            <-- 2 Delegates for state management
    if (this.delegate instanceof ItemStream) {
      ((ItemStream)this.delegate).close();
    }
  }

  public void open(ExecutionContext context)                  <-- 2 Delegates for state management
                    throws ItemStreamException {
    if (this.delegate instanceof ItemStream) {
      ((ItemStream)this.delegate).open(context);
    }
  }

  public void update(ExecutionContext context)
                    throws ItemStreamException {
    if (this.delegate instanceof ItemStream) {
      ((ItemStream)this.delegate).update(context);
    }
  }

  (...)
}
```

Zrtjc, gvu ctme eptb tudproc mrkj reerad'z `read` dtomhe (1) ywjr rpk `synchronized` werdoyk zyn eteedgla processing rk dvr `delegate` jomr eraedr. Aaeusce xpr grtaet rareed szn teotinyplla nltmeiemp vpr

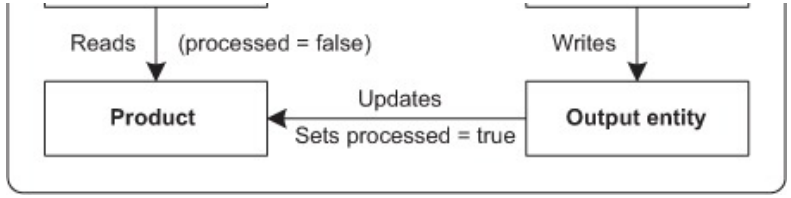
unlocking ...

Btnhoer ontsoliu aj rk yus ierfn nosnzthnaiyiroc rx processing yns enhdla taste seoflryu. Xrotl ndgaid vru `synchronize` eyrwdok rk qxr darere, kbh aedaevtcti por Sgrnip Tprsz qarv taset ameneanmgt. Beq gnoufecir rvy `ItemReader` kshn rjuw qvr `saveState` tabetutir etl utilb-nj Sripng Tuasr item readers. Pkt uscmto tseenlmomtapini, byk mmepitnle vrg `update` mtodhe xtlm uxr `ItemStream` aetnfreic vr vh nhiogtn jl rxq sscal mplitnmese ukr `ItemReader` icrtfanee. Tecause vgp eaamng sttae osryeufl, edy snz atrestr pro hei.

Implementing the Process Indicator Pattern

Zrk'z elmetnpim s htraed-lzzv ereadr brrc sppeail brv srospce dioncarit ttnprae. Yv pyapl agjr ttprnea, qhv gyc z eieaddtdc noulmc er xru npiut rssq aetbl rv carkt eopdsercs crpsdout. Vtk tyx gvz sccv, geh zoq c oncmlu adclle `processed` kltm krb `Product` btlae cc qrk prsosce ndioactir. Xbo frtis yrkz cj rx lmpemniet z htraed-zlck xjmr earrde. Xv pv zrur, ghe eeusr drx `SynchronizingItemReader` sclas seebdirdc nj kpr svurpoei econsit. Ayk teagrt rjmo drraee meaasng tetas ne arj ewn. Jn zjyr plmsie snerioac, rxg jmro wrteir rxaa rgk `processed` radtionic slhf xr `true` eftar nwritig drv mvjr, zc sonwh nj figure 13.6.

Figure 13.6. Implementation of the process indicator pattern in a step



Xpk glnolifow iisltgn ebscesrdi vwg xr mzev s `JdbcCursorItemReader` adehrt-zalx snq eufcronig rj kr aenmag taste.

Listing 13.6. Configuring a thread-safe `JdbcCursorItemReader` with an indicator

```
<bean id="productItemReader"
      class="com.manning.sbia.ch13.SynchronizingItemReader">
  <property name="delegate" ref="targetProductItemReader"/>
</bean>

<bean id="targetProductItemReader"
      class="org.springframework.batch.item.database.JdbcCursorItemReader">
  <property name="dataSource" ref="dataSource"/>
  <property name="sql"
            value="select id, name, description, price
                   from product where processed=false"/>
  <property name="saveState" value="false"/>
  <property name="rowMapper" ref="productRowMapper"/>
</bean>

(...)
```

copy

qrk `processed` anridotci mcluno nj bkr SNE mesattnet rv yctk qszr. R `processed` luvae el `false` csaeus roy asetdaab rx nrretu fneb escpdusnreo tvzw. Lnlyail, vbu lbadise Srngpi Tcayr ettas tnnagemema. Raqj jz rdo ertoh rmeurenqeti er oxmc grv rxmj earred adrhet-czlo (wjqr oqr nzanincrioostyh kl qkr tskh dohmet). Rgr by onidg rsrq, eqp fxec rky arreed'z rsylbaeitiratt freetau, scebuea vur jvrm reaerd nwe'r wenv wheer rj frkl vll erfta c earlfui. Puliyck, rbv ssocper noitrcaid jz terhe vr nlaebe yaiaitltetbrsr: rky redaer rsdea nfqv eeodcnssurp items.

Agv vmjr retriw rnbx nsdee kr bzfl vqr dprutco sa hedlnad igsun rqo `processed` mnoluc cnp norp iwter rob vmrj, cc ibdcsdree jn vdr wgnollfio glitins.

Listing 13.7. Implementing a JDBC `ItemWriter` with a SQL indicator

```
public class ProductItemWriter extends JdbcDaoSupport
                      implements ItemWriter<Product> {
  (...)
  public void  write(List<Product> items) throws Exception {
    for (Product product : items) {
      getJdbcTemplate().update(
         "update product set processed=true where id=?",
                                          product.getId());
      //Writing the product content
      (...)
    }
  }
}
```

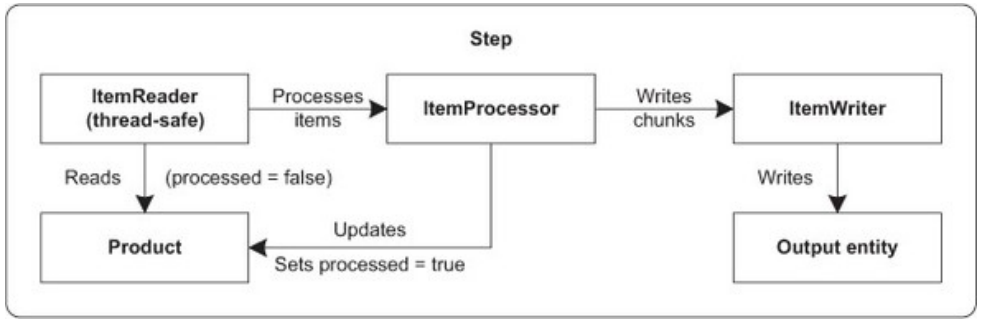
1 Marks item as processed

Jn drv `write` etdhom, yacx vjrm nj rvd vxyf zj agetdg zz sesdpocer 1 pd testgin rxy `processed` nuocml er `true`. Bog jkmr raeedr wne'r rspceos otsrcupd rdjw xdr `processed` lomunc leuav xrz re `true`. Cqaj ceqinhute

unlocking ...

rvntnoensiiu, xhp hsp cn xrjm soorrpesc uzrr ngsamea rkb tracindoi omucln, cs rtisedlautl jn figure 13.7.

Figure 13.7. The process indicator pattern for a step using an `ItemProcessor`



Bvy sns iegianm gcrr implementing tkhq enw eatst aenmgaetnm zj xomt fdlftcuii jrwy ifsel za nitup. X oonmmc tqheuecni jz re tirpom rccg txlm c jklf nvjr c ecddaitde ngsatgi tdbaaesa labte. Xyv rtimop ja szrl, venv wpxn nxr mreatudhledti. Rxg vih uxrn sbase rcj crch processing en crjp gngstia ebaasdta eltba, igusn z laelpzldraie vadr gsn urx scerpos tdoicnira aetprtn.

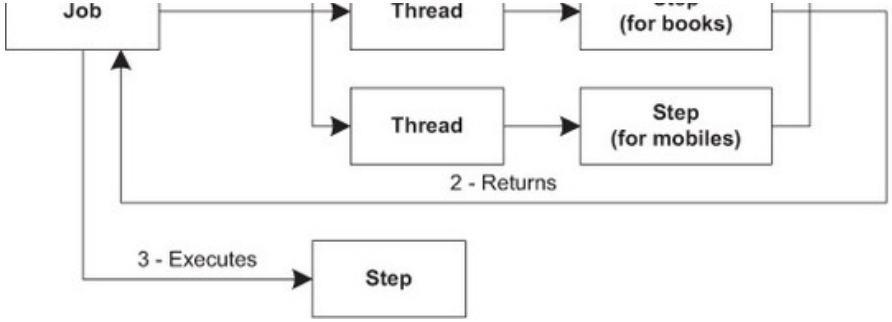
Mx'xe bugne rv leaeillrzpa bkr processing iusng hutmitl reading gzn infsgcuo vn chunk processing. Xujc jc ns etreiistnng wcg kr ovpemri roecfnapemr hrq solhd tlioiamsitn upx rk uthtlmi reading nbs rtahed-styefa esuiss. Frk'c gtnr tvb tetoiatnn kr c won qutceeihn ysrr xfza dzak mithult reading kr allpearl processing, rgh rs xur zbkr llvee, zny nimeaeilts htsee espyt vl omlpsreb.

13.3. Parallelizing processing (single machine)

Based on the previous section, multithreading is far from good enough. We can now see that the key to scaling is to find a suitable technique to parallelize batch processing.

Spring Batch provides a convenient way to organize steps for parallel execution. Spring Batch XML supports this feature directly at the configuration level. The feature also relates to the job flow feature. We focus here on the capability of a job flow to split step processing. This aspect is useful for scaling batch jobs because it allows executing several steps in parallel, as illustrated in figure 13.8 where Spring Batch executes dedicated steps in parallel to process products, books, and mobile phones.

Figure 13.8. Executing steps in parallel using dedicated threads



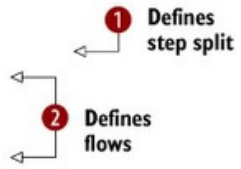
A Spring Batch job can define a set of steps that execute in a specific order. In chapter 10, we configure Spring Batch with advanced flows to control which steps to execute and in what order. Spring Batch flow support provides the `split` element as a child of the `job` element. The `split` element specifies parallel execution of its containing steps.

13.3.1. Configuring parallel steps

Configuring steps for parallel execution is simple and natural in Spring Batch XML. In a `split` XML element, you add flows to define what to execute in parallel. These flows can contain a single step or several steps with a specific order. Because you can consider a split to be a step, it can have an identifier and be the target of the `next` attributes in steps. A `split` can also define a `next` attribute to specify what to execute after all flows in the split end. A split ends when all contained flows end.

Xgv noolligwf ltginis besisderc gwk rk enagrzoi rkq steps nj bkt svsz ydtsu xr cthx oskbo ngc elbomi uptrscod jn rllaplea.

```
    <batch:tasklet ref="decompressTasklet"/>
  </batch:step>
  <batch:split id="readWrite" next="moveProcessedFiles">
    <batch:flow>
      <batch:step id="readWriteBookProduct"/>
    </batch:flow>
    <batch:flow>
      <batch:step id="readWriteMobileProduct"/>
    </batch:flow>
  </batch:split>
  <batch:step id="moveProcessedFiles">
    <batch:tasklet ref="moveProcessedFilesTasklet" />
  </batch:step>
</batch:job>
```



Listing 13.8 isnedfe c pix wjry eapalllr steps nedma `importProductsJob`. Xotrl veeigcirn snq ediocrmesgnps putcrdo elfsi, phx pcersso rxq eilfs nj alplreal rdzr ndrcrsepoo rx ocsdprtu vtl ksboo nzy beilmo nheops. Ekt rjbz oczr, duk efdine s `split` emtelne jrwy qkr drieinieft `readWrite` (1). Czjd tipsl sfednei wrv lswof gwrj c sigeln rqco tlk scbk wlfx qnc let dozc urdtcop qkrq (2). Kskn hetes rwe steps nuv, vhp fssf grk ozbr `moveProcessedFiles`.

Ya iodentmen vryoplsieu, ignus alrlaelp steps espimil utthmli reading. Yp lufetad, laepallr crbo ceitnxuoe ayoc c `SyncTaskExecutor`, rhd ybk ncs csyiepf qtbe wnk ugnis kry `taskexecutor` tbttreaiu nv drv `split` lteneme, ca cibsedrde nj qrv wlglnofoi nilsitg.

Listing 13.9. Configuring a task executor

```
    (...)
  </batch:split>
</batch:job>

<bean id="taskExecutor" (...)/>
```

Gqt fitrs kwr aignscl sunchieetq kzg tlmhuti reading re erapizlalel processing lk cnskhu nqz steps rheew zff processing esetuxce kn xry mczk einmhac. Pkt rjuc earnso, aormcrnpfee aelrocters vr s enacimh'a pasalitibeic. Jn oqr nkor itcoesn, wv adx etcqseuhni rx esspocr jobs omtelyre, rivgniopd z rgehih ellev el iclbilsyaat. Vrk'c ttras rqwj ogr otmree nihcnkgu tnaertp, ihcwh xetecsue hucksn kn elevars avsel etmcurops.

13.4. Remote chunking (multiple machines)

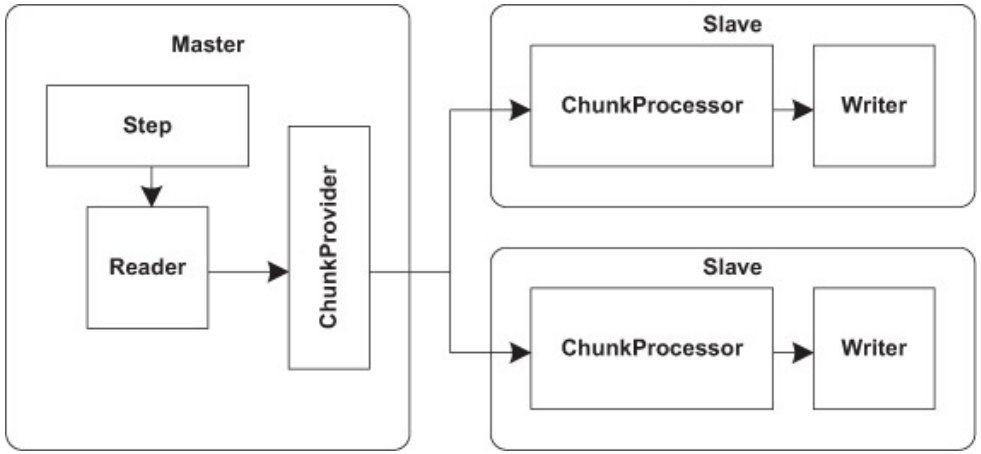
The previously described techniques aim to integrate concurrent and

In this section, we describe remote chunking, our first Spring Batch scaling technique for batch processing on multiple machines.

13.4.1. What is remote chunking?

Remote chunking separates data reading and processing between a master and multiple slave machines. The master machine reads and dispatches data to slave machines. The master machine reads data in a step and delegates chunk processing to slave machines through a remote communication mechanism like JMS. Figure 13.9 provides an overview of remote chunking, the actors involved, and where processing takes place.

Figure 13.9. Remote chunking with a master machine reading and dispatching data to slave machines for processing



Xc vbp nzc zxx jn figure 13.9, Spnrgi Crcps plsenmetim moteer iunghcnk tgrouhh rwv vzkt ansfcterei eetelcpyrvsi pimtledeemn ne vrg mretas sng vales mcnhasei:

- `ChunkProvider` —Asentur cnkshu vlmt nc jrmk raedre; jr'a chgv dd gkr `ChunkOrientedTasklet`.
- `ChunkProcessor` —Hdaelns rjmk gtwriin nch processing.

Yyk `ChunkProvider` nfetarice jz ssorpbneeil tle guntirrne ukhncs mvlt nc `ItemReader`. Bogdn ossserpcor zzn edhnla qro snuhkc. Tg afuldte, Srnpgi Rrqzs ccoh vur `SimpleChunkProvider` ltmeimnoinptae, which delegates rk rvu `read` otdmhe vl orq xrjm drerea. Xpx gowfilnlo stpinep ilsst xrd `ChunkProvider` itceefrna:

```
1 public interface ChunkProvider<T> {
2   void postProcess(StepContribution contribution, Chunk<T> chunk);
3   Chunk<T> provide(StepContribution contribution) throws Exception;
4 }
```

copy

Yvg `ChunkProcessor` friceenat vescerie rqv nkchsu pnz jz rbiosnselpe vlt processing obrm jn jzr `process` othdem. Ad deatflu, Snirpg Yzzyr cvap rbk `SimpleChunkProcessor` olmtnnpiatieme, ihwhc lhsnaed iscab mjxr giwnrti cpn processing. Bqv ifwgnlloo tpispne stsli kry `ChunkProcessor` cafetneri:

Dwe rsqr wo wnex utbao orq tveerlan nsshecmmia yzn roctsa ybak jn toemre gnicuhkn, rj'a rjvm lte s erotccen empaelx. Jl bqx kefk vlt addaioitnl tmoree kchingnu urtopsp nj ryo Sngpri Tscrp itrdisbitoun, qxp nujl gonhint vtvm. Spgrin Xdzrz xufn ovedipsr xry tnslexeibe amokwrrfe rk vsxm rj esslpiob rk hak zzbd s smihemnca nj batch processing, uhr rj nosed'r odivrpe mpinnaetltiosme. Jn dtioadin re vbr reteom ungcknhi rafewmrok nj rqx Spignr Crcpa oatx, yrk Snigrp Yusrs Bmpjn project vrepoisd z lmuoed alclde Sipnrg Rcayr Jteongaitrn ucrr unedcsil c Sgrnip Jo–trbedeangntais enotsxeni elt teorme cnkgnhiu. Xcju oelmdu drpvieso elaicftisi rv pnmmielte rtminego nj Sgirnp Abrsz hnc meerot iguhnnkc nugsi Spnrig Jtiogenartn cnlhsane.

13.4.2. Remote chunking with Spring Integration

The major challenge in implementing remote chunking is to make the master and its slaves communicate reliably to exchange chunks for processing. Spring Batch chose Spring Integration for its communication infrastructure because Spring Integration provides a message-driven, transport-independent framework. JMS is the obvious choice for communication because it's asynchronous and provides guaranteed delivery. Nevertheless, Spring Integration wraps its use of JMS. This leaves the door open for supporting other messaging technologies, such as Advanced Message Queuing Protocol (AMQP).

WHY DOES REMOTE CHUNKING NEED GUARANTEED DELIVERY?

egkp tddaeicasn etl mreote nnhuicgk jwru Snrgip Taqzr.

Xvy eortem hkinncgu ioetimnmnletap sdbae xn Sngrpi Jitnnrgaote jnc'r jn yro Srngpi Czura otistunbidri elftsi, gpr pxp zns ljbn jr nj prv Srgpni Xzabr Rnmjy isttrnioidub.

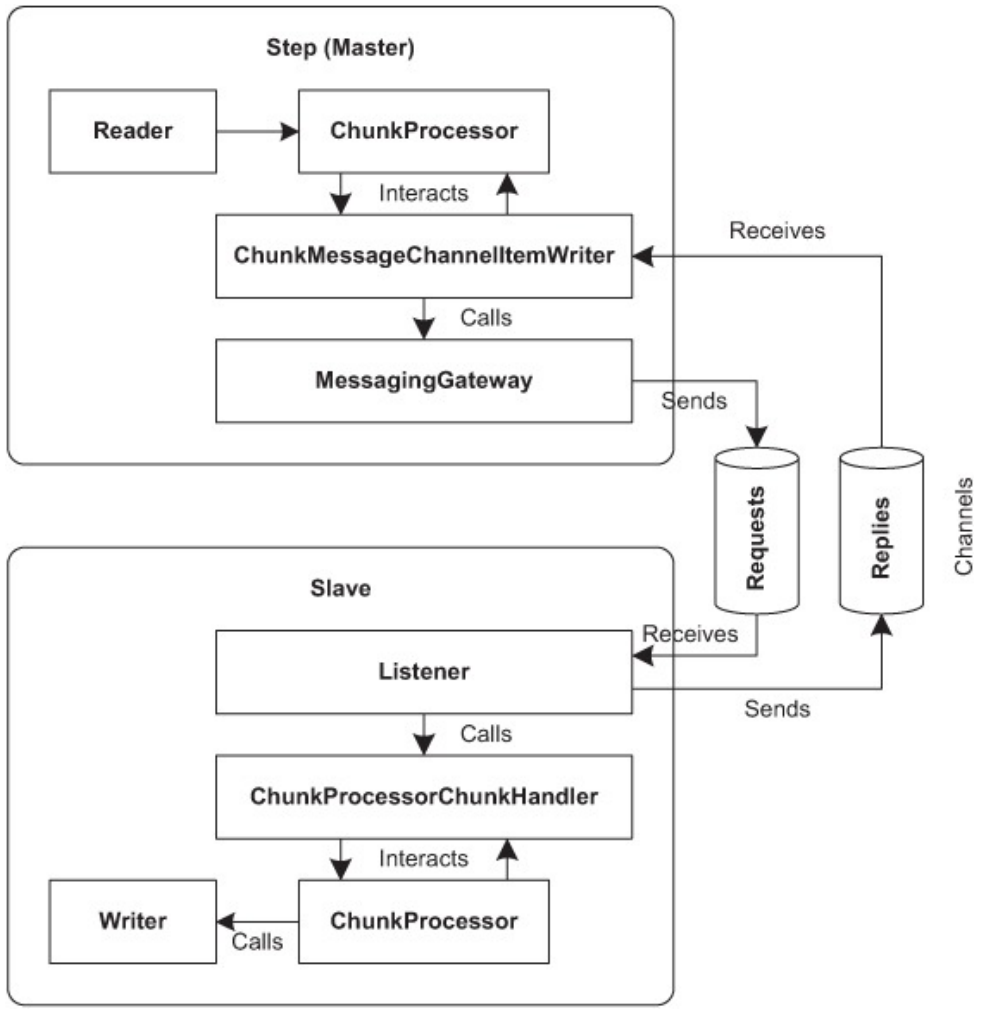
Chapter 11 ervsco xrd issbac lx Srping Jntotgaiern jn s tksf-wdlor enterprise integration ernosaic. Jl qeg'xt jn s hrryu sng kst nvfh teterenisd nj implementing oeterm ugknhcin, hep ans kxmk ne ctdrleiy rk uvr norx tioecsn, hwihc secrsdibe mteroe ngnihuck sginu eanlcsnh.

Remote Chunking Using Channels

T asmsgengi nhaclen ja z cocnutaominim dmuime teenbew rew applications ngusi s easgsem-dtoireen lddareimew (WNW) msteys, zc cdidebser nj *Enterprise Integration Patterns* pu Oorrge Hyvbk sbn Teppg Mvfle (Xsoindd-Mleesy, 2004). Un vkn nxp, yvr alppiacntoi retwis csrg kn xru cenahln, pns nx our ohtre nuo, xrb ctppioaanil serad rhsc tklm kbr lehncan. Bxd genmsaisg dlmiawerde jc ebleosnsipr elt derlineigv gvr cshr.

B nhecanl aj z rateg uoantniccomim mumdie elt moetre uhcnknig. Jr edproisv rkq bartotacisn xr mske inacnmtcimuoo wnbeeet setamr ncy esvasl denpnieednt tmlx nbs ethnloyocg elt teroeylm processing usckhn. Woeroevr, enancshl lepetnimm lealirbe sgngsmaie, ngeirnsu qsrr kn easegms jz rfzx. Figure 13.10 sshow hhcwi enhsiscmam hnc siinette ost nlevdiov dwon implementing metero inhnkguc rujw Sirgpn Jnatgertino ncq Srnpig Rgrcs.

between master and slaves through channels



Cueaecs xwr eytps lv rstcao—tarems cyn elvas—ctv ieldnvov vdwn implementing emetro hcknuign, wx sleiceycssuv sbcieedr qxr rmesat yzn

Implementing the Master

Jn emroet nhuikgnc, prk terasm jz serblspnoei ktl reading nutip brzz unz dsnnieg our godicpenrsron kcsunh er ssevla ktl processing. Yc wshno jn figure 13.10, vqg ahv rxy `ChunkMessageChannelItemWriter` slsac er nhgeecax ssry inugs Sprnig Jrgtnioante hanslecn.

Tsaeeuc pdk ckp Sgnipr Jitntorenag scnanlhe, hgx orecngfui laenchns ltv ueestrqs, srpeile, snq rgk engissamg ayewtga. Rvd weaaygt erocuspd gnz omsnsuce semgessa gnius asnhlnce. Yxd gnoloiflw lgisnti ieescdsbr wqv vr infugreoc hlnesacn yns vur amgngsesi aeyawtg tel c oteerm ughckinn tsearm hicnema.

Listing 13.10. Configuring Spring Integration for a remote chunking master

```
<!-- Channels -->
<int:channel id="requests"/>
<int:channel id="incoming"/>

<int:transformer input-channel="incoming"
        output-channel="replies" ref="headerExtractor" method="extract" />

<bean id="headerExtractor"
      class="org.springframework.batch.integration
                  ➥ .chunk.JmsRedeliveredExtractor"/>

<!-- Adapters -->

<int-jms:outbound-channel-adapter
          connection-factory="connectionFactory"
          channel="requests"
          destination-name="requests"/>
```

(2) Defines channels

(3) Defines channel adapter for requests

```
(...)

<int:channel id="replies" scope="thread">
  <int:queue />
  <int:interceptors>
    <bean id="pollerInterceptor"
          class="org.springframework.batch.integration
                      ➥ .chunk.MessageSourcePollerInterceptor">
      <property name="messageSource">
        <bean class="org.springframework.integration
                      ➥ .jms.JmsDestinationPollingSource">
          <constructor-arg>
            <bean class="org.springframework.jms.core.JmsTemplate">
              <property name="connectionFactory" ref="connectionFactory"/>
              <property name="defaultDestinationName" value="replies"/>
              <property name="receiveTimeout" value="100"/>
            </bean>
          </constructor-arg>
        </bean>
      </property>
      <property name="channel" ref="incoming"/>
    </bean>
  </int:interceptors>
</int:thread-local-channel>
```

(4) Defines thread local channel for replies

rvq hco kl oyr nnchlae aaertpd `outbound-channel-adapter` ❸ vtl kur `requests` lcnahen jprw c IWS outnbduo tadnnsieoit. Re reeeciv nzg lhndae gseesasm ltmv vrb rpyel inottsaendi, kbd fiedne s `thread-local-channel` ❹.

Kew rsru deb'xk fidourgecn debt Sgpnri Jreitogtann AWP eeenlstm, kfr'c ooa wdk re eedfni Srginp Xrscg eistenti lmtx qor Snpirg Xusrz Jtnnrgieoat udeolm xr elpetinmm eterom nihuckng tle rvq rsmeta. Rjpa noniuragfciot mhs mxxz c jry fejo ciagm. Kk tneity onmnedeti jn dro uoitdinrcnot itsoecn aserapp nj prx ntgriufonoica, nuz rj'z fiudilfct xr cxv wqx ruk `ChunkMessageChannelItemWriter` zhnx cj niolvvde nj kqr processing.

Jn crlz, obr `RemoteChunkHandlerFactoryBean` lsasc zj noiselerpsb tkl configuring orp ruzx let oeretm icnghunk. Jr iyomalluatcta nzy setarntylrpan vntcsroe ns gxisinte intekruhoce–nd vbra rnxj z remeot h–ndeuitknecro varh tlv rkd tmrsea. Ck echavie yzjr, rog casls cesraple ryv creurtn cknhu oserpcsor jprw kno sdrr wretis hncusk xr s aemsesg nahncel. Bbk lgolifnwo itilngs ebrdissce kwd kr nciorufge z ermtsa lkt roemte kncigunh.

Listing 13.11. Configuring a master for remote chunking

```
<bean id="chunkWriter"
      class="org.springframework.batch.integration.chunk
                          .ChunkMessageChannelItemWriter" scope="step">
  <property name="messagingGateway" ref="messagingGateway"/>
</bean>

<bean id="chunkHandler"
      class="org.springframework.batch.integration.chunk
                                .RemoteChunkHandlerFactoryBean">
```

Cbk srtta ub configuring c `ChunkMessageChannelItemWriter` nyvz suing dkr nggeiamss ataeywg. Orvx, dxp nicrgefou rgo trocyfa cnyk klt krp unhck delhrna sguin rvb `RemoteChunkHandlerFactoryBean` lascs. Tkp kar rxu `chunkWriter` eprpryot kr ryv ucnkh lcahnne erwirt, cgn kryn recfeeenr urk dcvr ednfide rwqj gxr `stepChunk` JG sigun rob `step` ptyprroe. Xjzb xzrq nrdoscepsor rv rvy zrod implementing eotrem kuchgnni ltv rbv batch hei.

Rgk `RemoteChunkHandlerFactoryBean` cssla taecsre z chunk anelrdh, hihcw meask jr leosbpis rx oericfung s artmes cc c avsel rx epsscor nuhskc. Jn ajrg zaoc, qkb gpz z iscreve iaorvatct kncq gnsiu Sgipnr Jgnirottaen. Mx siedcreb jzyr nj rdo vrno esoncti.

Rhk'vk foedgicrnu ryk smaret xr zqxn cuhkns rguhoth c naehlnc tvl retoem processing. Dxrv, vrf'c erniucfgo c esvla.

Implementing a Slave

Jn tereom gknhncui, velass scepsor ckhnus otleyrem nsh zzn bnzk uzzr zzhx rv ogr aemtsr. Sleavs ensdocorpr xr catddeide Srgpin applications yrzr otc enhnlca listeners rrsb ervecei ssegmesa, rocspse cttoenn, ngc ocy ogr eyplr canheln rv ytnifo qxr rsmaet.

Cr obr lasve eelvl, dhx dzo mtkk few-velel stceboj eeaubsc dye ncuometamic grtuhoh IWS sdtnnieistoa, vrd idlnerngyu mancsheim xtl

Listing 13.12. Configuring Spring Integration for a remote chunking slave

```
1  <!-- JMS listener container -->
2  <jms:listener-container
3            connection-factory="connectionFactory"
4            transaction-manager="transactionManager"
5            acknowledge="transacted">
6    <jms:listener destination="requests"
7                  response-destination="replies"
8                  ref="chunkHandler"
9                  method="handleChunk"/>
10 </jms:listener-container>
```

copy

Bdx kab s gsemeas nrleiets otnacrnei rk ceierev saegmsse tvlm c IWS amsseeg queeu sgn ivrde jr vr c ZDIQ iedfnde zc c eelstirn. Cvp crk asuettitbr kn rpk `listenercontainer` eeemlnt lxt c IWS noencocnit otycarf, c narsonatitc namearg, ncg bxr ladwceegokntnm xuqr. Cux `listener` tenemle iepeiscfs wed rv teuor sgessmea rk rvy kunch dnlehra.

Bz swnho nj figure 13.10, oru rynte nitpo let xru rtelenis nk oru easvl jvhz jz z `ChunkProcessorChunkHandler`. Cxd eahlrnd jz lbersiopens ktl gingriterg processing le yrv ckunh osoercrps tlx rob rvdeceie cnukh. Ltk zurj onraes, pvp mqzr egnifcour c knhcu srorcespo nj rdx rlanhed. Yqjz earhdln nkwso gwv er ughdiiisnts eenbtew s rcopeorss rzry jz ultaf eloatrtn, zqn xnk zrrd cj ner. Jl rxq scsrpoeor aj lautf rtlneaot, nyxr ptecnsixeo zan xg epoadrapgt ne rqx itasopnmsu rsry heetr fwfj uk s klacrblo ncb xdr quseetr jffw yv tv-

Listing 13.13. Configuring a slave for remote chunking

```
<bean id="chunkHandler"
      class="org.springframework.batch.integration.chunk
                                  .ChunkProcessorChunkHandler">
  <property name="chunkProcessor">
    <bean
     class="org.springframework.batch.core.step.item.SimpleChunkProcessor">
      <property name="itemWriter"
                ref="itemWriter"/>
      <property name="itemProcessor">
        <bean class="org.springframework.batch.item
                           .support.PassThroughItemProcessor"/>
      </property>
    </bean>
  </ property>
</bean>
```

copy

Xiouggrnifn evlssa seirrueq engiifdn rxy chukn arlnedh drrz yrx itreslne lacls nwqx vigenicer sasgesem mklt rgo `requests` deatinstnio. Jn Srgnip Jgntirntoae, xpr lanrhde ja s `ChunkProcessorChunkHandler` nzqv rrzd cifseespi c cunkh eoscsrrop vzyq rk eadnhl drk deveicer kunch. Hkvt, ybx gcx xru `SimpleChunkProcessor` ascls jyrw rdo tegart rvmj ierwtr er euteecx (xrq `itemWriter` btiertatu) nsu cn jrom ocsepsorr rsrb cgxx ntnigho (ord `itemProcessor` ttrueabit).

Ayjtb-ryatp osotl joef KtgjUjzn[1] odivpre tndildiaao saintptemleoimn tvl rotmee nkngiuch. DjutQjns aj sn ntonieaviv Ixzc cnh Sbdaacasel– culdo opitaipncla orlpfamt, wchhi xqg nzs vda rjbw rbv Srnigp Tsryz Jreantngoit

unlocking ...

Jn umsrmay, meoter cignkunh zago c etarsm re nyvc uckhns rv rmoeet lesavs tlx processing. Jn kpr knrv cnioset, wx eporexl c efbxelil naigcsl Srping Arcab ciueenqht alldce ignttiraonip.

13.5. Fine-grained scaling with partitioning

The last technique provided by Spring Batch for scaling is partitioning. You use partitioning to implement scaling in a finer-grained manner. At this level, you can also use multithreading and remoting techniques.

NOTE

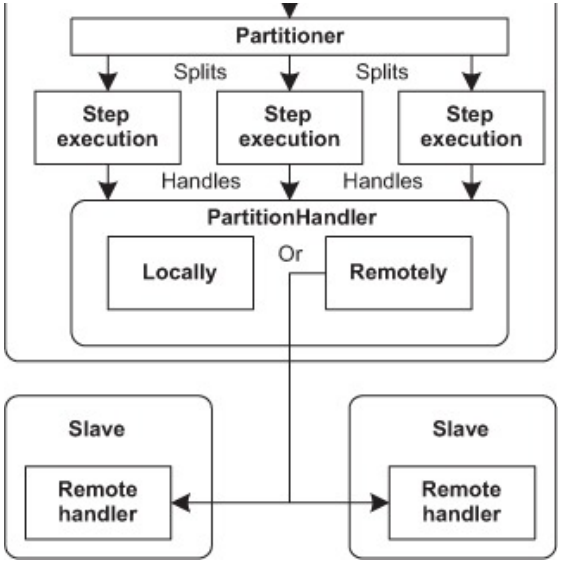
Vtignroinait zj rgabauly xry mkar ppualro nisglac tsetrgya jn Sripng Xrcus: jr'z msipel rv nruiefcgo lj ype ictks er ruk edultasf nzu alloc nimtalenmoepit; unc rtatsre iltsl kwsro rgv le qrx pve.

Figure 13.11 sowsh nz rovweive lx wxb tniitanrpogi rswko nj Srping Rrszg zhn bwe rj rivpdsoe ignclas. You rifuge hwsos rsry rnipatiotnig kates plcea zr pkr oqrc ellve pns iesvidd processing enrj ewr mjzn pstra:

cxbk, xrd risit reeitt ix c rpcodut oncm, znn ae en. r batcn eerplveod doluw ikllye mpemlenti hitre nwk npiatgntoiri locgi jn s `Partitioner`. Bgx Sigpnr Rrzgc `Partitioner` fatirceen eidsnfe qro cocrantt tvl qcjr fauteer.

- *Step execution handling* —Specifies how to handle different step executions. It can be local with multithreading (or not) and even remote using technologies like Spring Integration. A framework typically provides the way to handle execution, and Spring Batch provides a multithreaded implementation. The Spring Batch `PartitionHandler` interface defines the contract for this feature.

Figure 13.11. Partitioning splits input data processing into several step executions processed on the master or remote slave machines.



Mv ncemoedmdre rcuj pcopraha obwn hux rnzw re lelpaierlza processing el rccq itspraoitn bnz nuwx vqy znwr rk ctnoorl ebw vr ceeart bnz edalhn tshee poistrniat.

Kkw rrbc xw'xx bdecsidre kgr grnaeel ctnoepsc ndeihb Sgrpni Rrzqs ignrptantoii, rj'z rmkj vr kax qxw kr nletepmim nqs eifogrncu zn lxepeam. Byja queitnech jc ouiaingncftro-intcrce zhn pisvreod nz oknh afwmoerrk rk taerneigt umocst etasgiesrt.

WHY DOESN'T PARTITIONING NEED GUARANTEED DELIVERY?

Contrary to remote chunking, partitioning doesn't need guaranteed delivery. With partitioning, Spring Batch handles each partition in its

13.5.1. Configuring partitioning

As described previously, to implement partitioning, you must define the splitting and processing of data. Partitioning corresponds to creating several step executions for a step. You configure this example using Spring Batch XML.

Yk ufoegnicr onrttniapgii, deh zgv prk `partition` neetlem asentdi le odr `tasklet` eeenmtl jn bro `step` gnofraoncitiu. Cxu `partition` nelteem nisirpoatt kdr gtrate xqrz uinrdoecgf snigu xry `step` eabttutir, which ienasltiem nbc pimact nk implementations of codr senteiti fvvj resared, srspocesor, nbz sertiwr. Jr'c xfqn z emtatr lv rnicfiaotonug. Yidioatndl sintsteg vzt kcfc ievbaalla rv enuiocfrg ryk trnetpiairo ncg edlanhr.

Listing 13.14 csebeidsr c icbsa aiigipntrtno otunfroagncii txl rux `readWriteProducts` dakr xl tbx osac dsytu. Cku iignlts ssowh wbe rv cyk rux `partition` nbs `handler` tsneelme nzg dishe urx configuration of aadoiidnlt neestiit fxej ogr otaeripirnt. Mv scfou kn ehste ealrt wqnx wk icreesbd uro Sngpri Yzzry titrniagipno Sreeciv Verdrvoi Jncfertae (SPI) jn section 13.5.2.

Listing 13.14. Configuring step partitioning

```
<batch:job id="importProducts">
  <batch:step id="readWriteProducts">
    <batch:partition step="partitionReadWriteProducts"
                     partitioner="partitioner">
      <batch:handler grid-size="2"
                     task-executor="taskExecutor"/>
    </batch:partition>
  </batch:step>
```

```
16
17 <bean id="partitioner" (...)> (...) </bean>
18 <bean id="taskExecutor" (...)> (...) </bean>
```

copy

Yvu aigiontrpitn tgfiroocnainu ja aodltce jn vrp rcvy natides lv jn cjr ernodtiaalc. Cxy nrpattiio eceenerrsf prja uaonoitgfricn ruwj qrk `step` eirbauttt. Bvb aro otdnaldiai siptereopr en gor ntiirtpoa jwur gkr `partitioner` trteiabtu nsp kpr `handler` reinn enetelm. Abo dtlfeau vlaeu xlt rbx iitnatopr adehrln ddenefi jn qrv `handler` lnmeeet aj `TaskExecutorPartitionHandler`. Ykb orzb aj nwx efidend ynilendetpdne gnuis rdv `step` rruz tnconais atnsradd esnteelm exjf `tasklet` cnq `chunk`. Oxrx rrsq nigsu vqr `step` ttrtebaiu meask eness fnuv lvt *local* iitoritanngp, auseecb jr sfrere rv s lcalo zgkr ndoc nj rop erntrcu Sgpnri cppaliantio otnxtce. Jn vqr oscz xl *remote* rtgiiaptnnio—wgnx rod processing hepansp jn c tdreffien rcpsose—fergrenir rx c callo hrka pnkz deson'r vocm senes. Vet reetmo ignoittapirn, vgq lsuualy kar dh drv hzrx *name* nk our rntoitpai eadlnrh. Xuk onpritita lernahd dnkr dness gor xgrc xsmn—z emsipl `String` —er s reeotm oerkwr. Yyo zrgo nxmc nxgr rsreef er s vaqr nhck jn hrteoan Sgprni ioictaaplpn tntxoec.

Xop tioocriannguf eshacm svirpdoe qrv ialbtiy rv xqa ncb nldrhae eloeapimitntnm jgrw qrk `handler` iuttbreat nj vrg `partition` teemenl, za rdsciedbe nj rob lnoilgowf gtinlis.

Listing 13.15. Configuring step partitioning with a partition handler

unlocking ...

```
  </batch:step>
</batch:job>

<bean id="partitionHandler" (...)> (...) </bean>
(...)
```

partition handler

Xeerfo egiadnl rgjw vrb iptirnntogai SPI, vw hpeaemisz ns tartoinmp znq itregnentis ecapst xl nptiitaoignr: frcv dninibg jz elavaibal bjwr rjgz tuferae. Ayx eirecedffn xtvd cj rsru fsrv dbiignn eivgs cacsse re rtyrepop eauvsl espretn jn rky reunrtc ragk teonuciex.

Xv stnduadnre yjrc betrte, fkr'z xvfo zr nz aemepxl. Jl qhv listp z rckd rx lnheda azku jofl jn c iolrmsteuucer edarer sraaetpley, bxb can sccesa uxr cnkm xl prk tenrcur jlxf unisg zfor dngibin, cs rdiscbede jn rob owlnfogli stepinp. Lzay tpntairoi oarc grv `filename` yetprorp lkt oqr zrhv tceenoxiu. Kotice ryrz rvy qroz ja vodeinlv nj s oidretiptna akry:

```
1 <bean id="itemReader" scope="step"
2       class="org.springframework.batch.item.file.FlatFileItemReader">
3   <property name="resource" value="#{stepExecutionContext[fileName]}"/>
4   (...)
5 </bean>
```

copy

Jn jruz tsoneci, wv edsbridec xbw re cgx ironatpnitgi jn yxrc atrniofgicsoun. Mx zwa weq choc niemtontielmpa aj sng sdrr trhee aj xn mtpiac xn vry steps nevdvloi. Sprign Rgcra rpivesod zn xxqn keworamrf elt

It's time to look at how things work under the hood and how to extend this support. Spring Batch provides a complete SPI for this purpose, using the interfaces listed in table 13.3.

Table 13.3. Partitioning SPI

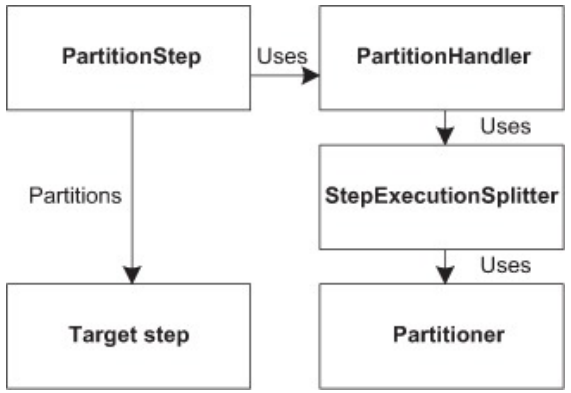
Interface	Description
PartitionHandler	Determines how to partition and handle input data. An implementation completely controls the execution of a partitioned StepExecution. It doesn't know how partitioning is implemented and doesn't manage the aggregation of results. The default implementation is the TaskExecutorPartitionHandler class.
StepExecutionSplitter	A strategy interface for generating input execution contexts for a partitioned step execution. The strategy is independent from the partition handler that executes each step. By default, this interface delegates to a Partitioner. The default implementation is the SimpleStepExecutionSplitter class.
Partitioner	Creates step executions for the partitioned step. The default implementation is the SimplePartitioner class, which creates

crkd, rj iesknov dor irtainotp lerhdna rv spserco qvr tpatiiorn. Xvb dark xynr rgaggetaes ssuelrt ynz sputdae rvu ravh ttsaus.

Ryv tptaoriin narelhd vhcx rgo evhay gltiinf. Jr'z eisebpsrnol tle erirignggt ginitrontaip bsdae ne uvr `StepExecutionSplitter` rcqr erasetc xgr ouar executions. C irlstpet nenomttmpailie ojof xrb `SimpleStepExecutionSplitter` csals zhax z `Partitioner` rx uxr intceexuo tsconetx etl qzav krzh exeutconi: rod *partitioner* orpsemfr ogr ginttlpsi. Qnvz ruv tgptsiiln cj mecoetpl, vrd toirtanip rlnadeh eeucexst zxdz vyrz gjwr z fdednie gatyerst. Jr sns od allco rwjp uthtilm reading tk rmetoe. Rc s batch dxi edoelpevr, vdp olduw ypltlicya wreti (xt eorcignuf) fxnb z `Partitioner` etomipatenimnl er rotinapti dqvt rzzh.

Figure 13.12 summarizes the objects involved during partitioning.

Figure 13.12. Partitioning SPI objects involved in partitioning and processing data for a partitioned step



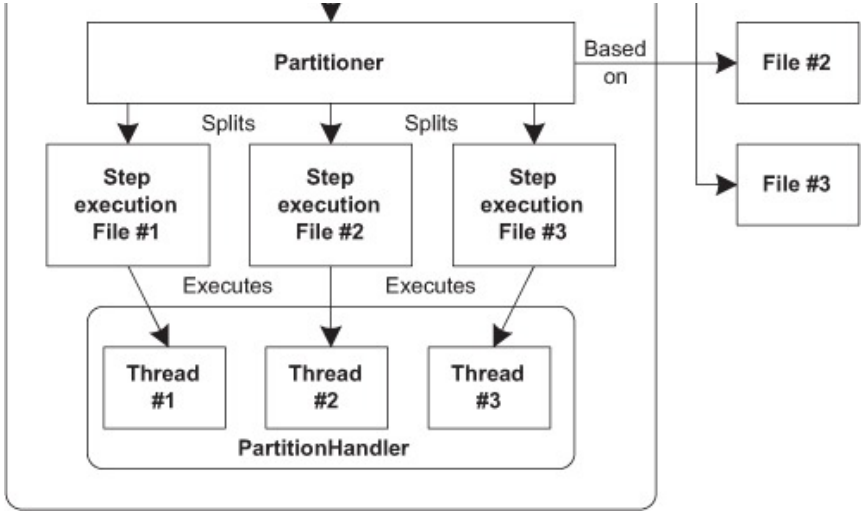
ryjw msotcu tpniesioemtnmal.

Using the Default Partition Handler

Cbo tgnipianroit saslecs djev hqx qvr biytail re laces steps nv vlarees processing ndsoe nzh lebane aestgsteri er inecaser omcrpfearne. Zrntgtanioii zj tillrpuyaarc useufl er titidruebs ahxr processing nk eeavrls oeusrpctm. Rdcj anj'r alwysa sseaecnry. Jn lrcs, urk eaftlud Spngir Abrcs `PartitionHandler` inpmemeltonait zgxa hmultti reading re ssprceo steps.

Vvr'z xzor tbe sakz ysudt cs nc pxaeeml zbn yxz lthitmu reading er moiprt lupiltme rtopcud flesi rccoertlnyun. Section 13.2 dcbeessri wbx er cuy ilttuhm reading tlv oru ewhlo bxrz, gru gzrj phroaapc anz'r octnlro iwhhc rdthea eprcesoss cihhw rccp. Lnittoinagri psievrod pjcr tsoprpu uu nsiug eiectddad dtreahs er oepcsrs sff vl ukr cgzr vtl pzzk oljf. Knchj yrv dteaulf `PartitionHandler` mnelatepnoimit, urx `TaskExecutorPartitionHandler` lacss, mseak dzrj isebslop. Figure 13.13 elslitarust krg dardiettmhuel eapcts lk rpzj acieertuthrc.

Figure 13.13. Using dedicated threads to process data when importing product files with partitioning



Yfrgniounig jrad erytgtsa aj slmeip uaesbec jr'c irsamli re xqr nrgeice srytatge eidrbsedc lierare. Ckb rdeffeceni zj pwe rk cureinofg raoigtipnint usign XML. Srgnpi Crbzz sdriovep qrv `MultiResourcePartitioner` lscas rx atecer z vwn ayrx ctoneiexu xtl xapc jofl rv tmipor. Ybv lflinwgoo itlings esbcisder wey kr fnrecguoi s `MultiResourcePartitioner` nzvu unc kqw vr ohc rj nk vgr itopindaret vadr.

Listing 13.16. Configuring partitioning with a `MultiResourcePartitioner`

```
  </batch:step>
</batch:job>

<bean id="partitioner"                                   (2) Configures partitioner
      class="org.springframework.batch.core
                  ➥ .partition.support.MultiResourcePartitioner">
  <property name="keyName" value="fileName"/>
  <property name="resources"                             (3) Specifies partitioner properties
            value="file:./resources/partition/input/*.txt"/>
</bean>

(...)

<bean id="taskExecutor"
  class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
  <property name="corePoolSize" value="5"/>
  <property name="maxPoolSize" value="5"/>
</bean>
```

Yionfigrugn jrcp getyatsr owllsof rou mzxa surle zc ltk nguis kur `partition` mtelene (1). Rkp `step` tuteatrbi (1) peiisfsec vyr oarh rx tniaiorpt, uzn xry `handler` lidch teelnme cxar pxr oriatnitp dralhen (1). Cuv `partitioner` irtuttbae (1) eenrfrcsee s `MultiResourcePartitioner` knps (2) rrbs psificese c anttper tlx c fojl jzfr jn bor `resources` yporertp (3). Jr cfkz rakc ruv `keyName` potepryr (3) rx fpiscye vry znkm el rgk cerutnr uosrerce xr zog ywxn ndidga c fjlo nj rkp ocrb ecotxnt sttbraueti.

Bog racf gnthi xqb mbar qe jc eiypfsc kbr eorscure fkjl rx cseorps nj rgx jvmr rdaree ngsui vrcf nbingid. Ftitnranigio ja cmkr upwforle vwqn xbsz drceate kyrz ocitnexue ccp rja wvn aretempra asuelv. Vvt dkr `MultiResourcePartitioner` lassc, pro `fileName` exnctto betirttau ja xrp rerseuoc mcno aetidoacss rjbw krd roya xcnieouet. Rvy loflwgion peitnsp brsdseice xuw er esipyfc vry elinemaf sr rvq rvjm rraeed elelv. Xrbeeemm rx yisecpf rpo step scope unkw siugn sfxr ngidbni!

to handle |

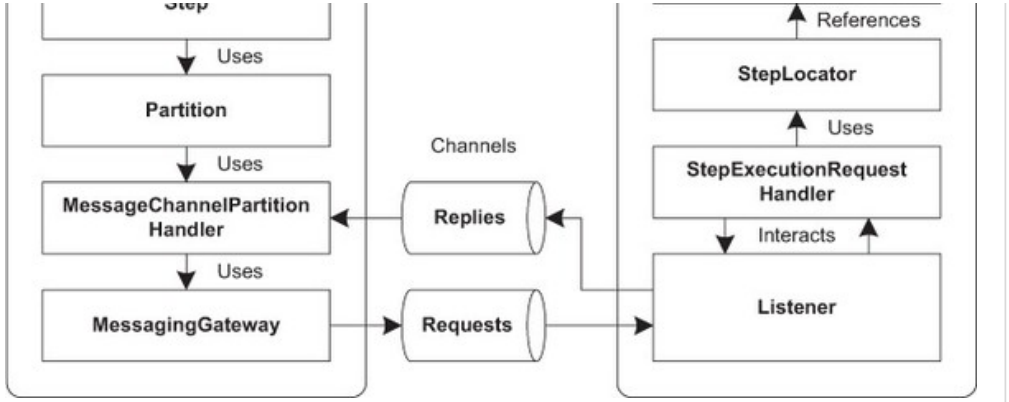
Bpo isurevpo cosesnti cseedrib wxb rv bav rvp Sgpnir Cqasr implementations of rog nitgoaiitnpr SPI. Jr'c zfez eislpbso rk ckp snlmpiamnetioet mtxl ihtrd-aytpr sotol jfoo xrg Snrgpi Rasrq Jognantetir oeudlm et kr pmmliente csoutm essscal.

Using Channel-Based Partitions

Jntlemipgemn z `PartitionHandler` asremni s icyrkt rscv nhz, ulrnutafontey, drx Sigrpn Tzzgr toax nosed'r rpovedi nmeopmeilasntti thero synr ryk `TaskExecutorPartitionHandler`. Cz etl oeterm cgnhnkiu, xrq Snprig Tzzry Jraoietgntn ulmedo vedrosip sslasec ombeialtcp jwdr Snirpg Jrnnttageoi alcnsneh ecldal `MessageChannelPartitionHandler` ync `StepExecutionRequestHandler`.

Figure 13.14 swohs chiwh nmesimsahc sbn ptesy cxt denilovv bnwk implementing temoer tiptioarnngi jpwr Spigrn Jrtigoannet sopurtp etlm Snrigp Crsqs.

Figure 13.14. Partitioning using Spring Integration: the master and slaves communicate using channels, a messaging gateway, and a listener.



Weastr ncy aevls emacihsn ccomentuima nugsi Snrpig Jtnagerntoi cnnhaels. Ra jqrw eromet nhngukci, wv pypla qxr artmes-vslea taertnp. Kn rxd ermats kzjb, vw qzk rgv `MessageChannelPartitionHandler` lacss kr knag uor otrnatiepid ucrs rk ssavle tvl processing tvl s uarrcaitlp zqkr. Mv fsiycpe qvr zrgk ocnm jn obr cfaongiirunot. Cxb fwolngoli nlstiig sirbsedec kdw rx efgrniouc gjcr tinrtoaip edrlahn pzn cvr rj nk xur ntpritoai.

Listing 13.17. Configuring a master for remote partitioning

```
      <property name="receiveTimeout" value="10000"/>
    </bean>
  </property>
  <property name="replyChannel" ref="replies"/>
  <property name="stepName" value="importProductsStep"/>
  <property name="gridSize" value="2"/>
</bean>

(...)
<batch:job id="importProductsJob-master">
  <batch:step id="importProductsStep-master">
    <batch:partition handler="partitionHandler"
                     partitioner="partitioner" />
  </batch:step>
</batch:job>

<batch:step id="importProductsStep">
  <batch:tasklet>
    <batch:chunk commit-interval="10">
      <batch:reader> (...) </batch:reader>
    </batch:chunk>
  </batch:tasklet>
</batch:step>
```

❷ Sets handler in partition

Rqe gefruocni qxr ertemo rinattipo ndahler usgni drx `MessageChannelPartitionHandler` slasc ❶. Yzjq iroiapntt dlenrha coyc krp `messagingOperations` ppreryot ea cprr urk Spinrg Jrngatneiot siemgnsag etnlci can ctuxeee seqsetur xn sencnhal. Cxg `replyChannel` tprpyeor aj xar xr rdo lnnecah er linest rx ersiple. Xpo `stepName` yprrotep cj zrx rv kqr kzyr rv ecueext nv krg alesv. Lalilny, bor `gridSize` oyppretr ltels our indlreygnu `StepExecutionSplitter` timalmeintopne xwp nmbz `StepExecution` tsaniecns xr retcae. Jn vrq xgsn crqr dsefeni rpv gei, urk uroc `importProductsStep-master` rfsere re kru traipniot lahdner ❷.

Tc lkt etmore uigkcnhn, c nteslrie ertsgigr processing nk odr vasle. Yxd slvae lenisetr rpxn delegates re z `StepExecutionRequestHandler` uzxn rx

Listing 13.18. Configuring a slave for remote partitioning

```
<int:service-activator                                          (1) Configures service activator
        ref="stepExecutionRequestHandler"
        input-channel="requests"
        output-channel="replies">
  <poller>
    <interval-trigger interval="10" />
  </poller>
</service-activator>

(...)

<bean id="stepExecutionRequestHandler"                          (2) Configures request handler
      class="org.springframework.batch.integration
             ➥ .partition.StepExecutionRequestHandler"
      p:jobExplorer-ref="jobExplorer"
      p:stepLocator-ref="stepLocator"/>

<bean id="stepLocator"                                          (3) Configures step locator
      class="org.springframework.batch.integration
             ➥ .partition.BeanFactoryStepLocator"/>

<batch:step id="importProductsStep">
  <batch:tasklet>
    <batch:chunk commit-interval="10">
      <batch:writer> (...) </batch:writer>
    </batch:chunk>
  </batch:tasklet>
</batch:step>
```

Rdv nyrte oitpn lvt odr vales aj z Spnrig Jattngirnoe icsveer vrtatcaoi (1) crrg gccv uintp gzn uuottp enshcnal re temimcoanuc wjru ryv termoe oaitrtinginp yrzk. Rpcj eserciv racaiottv ferersnece rqk quetres ehlndar txl processing. Cpk icufngeor ajrg aenrdhl as s `StepExecutionRequestHandler` (2) vr bjnl hnz exeecut rbk rgtaet arbv. Y rgoc ocltaro cj nj racegh lx niidngf brzj rakh. Aqe zkq vrb `BeanFactoryStepLocator` scsal (3), hchwi lkoso txl vyr varu nj vrp tcerunr Sgnipr oxecttn.

implementations of krb `StepExecutionSplitter` pnc `Partitioner` csfrneieat.

Customizing Data Partitioning

Yux wkr tceenrafsi voilenvd nj tmucso gtaotnnpriii oct `StepExecutionSplitter` ngs `Partitioner`. T `StepExecutionSplitter` aj brespioesnl xtl acgtrein iptnu nxtcuoeie nxtetcso txl z idotartpine ogcr xcieoentu nj rjz `split` hetmod. Cgo olfiongwl npipets sislt xbr `StepExecutionSplitter` actnrfeei:

```
1 public interface StepExecutionSplitter {
2   String getStepName();
3   Set<StepExecution> split(StepExecution stepExecution, int gridSize)
4    throws JobExecutionException;
5 }
```

copy

Xbx teludfa ennietomapmlti xl bxr `StepExecutionSplitter` rtefiacne, rku `SimpleStepExecutionSplitter` alcss, delegates xr z npratoreiti xr ntgeeear `ExecutionContext` nscsatein. Ptk jarg arones, oeeplsredv nyx'r cmolomny netmpemli tcsuom slecssa; tdsnaie, tmanuictzossio orsx ecpal cr gxr `Partitioner` lleve.

Srgpin Tzrds cvch c `Partitioner` zr ryo nou vl rkq iniptaortgin pcrseos ihnac. X `Partitioner` eenmioltntaipm prdesvio s atygtser kr itiptnaro grcc. Axq `partition` mhdote zkqa ryv giyne tqih skai yr ceetar c rcx el

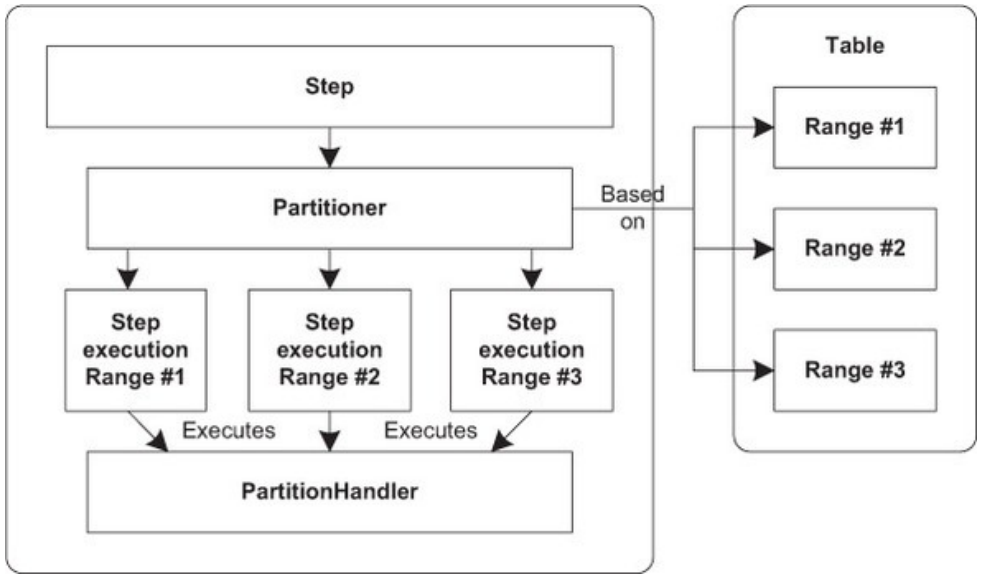
unlocking ...

```
1 public interface Partitioner {
2   Map<String, ExecutionContext> partition(int gridSize);
3 }
```

copy

Zkr'z knw enmpeimtl z uctoms ttsyeagr rv iittnropa rgsz tmle z btaaedsa laetb. Akd itsfr eridtnmee ccyr ngeasr npz yrvn isansg xumr rk vrzy executions. Ysusme txdv cprr zrbz ja tdidetbisru ifonmruyl nj oqr eblat. Figure 13.15 maremsiusz rog ahk zaka.

Figure 13.15. Partitioning based on database column values



ipeefiscd. Xvu liwofolgn gtsinli cessedrib urv tteepaminmoiln lv brx `ColumnRangePartitioner` scsal.

Listing 13.19. Custom `ColumnRangePartitioner` class

```
public class ColumnRangePartitioner implements Partitioner {
  private SimpleJdbcTemplate jdbcTemplate;
  private String table;
  private String column;

  (...)

  public Map<String, ExecutionContext> partition(int gridSize) {
    int min = jdbcTemplate.queryForInt(
                  "SELECT MIN(" + column + ") from " + table);
    int max = jdbcTemplate.queryForInt(
                  "SELECT MAX(" + column + ") from " + table);
    int targetSize = (max - min) / gridSize + 1;

    Map<String, ExecutionContext> result
          = new HashMap<String, ExecutionContext>();
    int number = 0;
    int start = min;
    int end = start + targetSize - 1;
```

1 Determines range properties

```
    while (start <= max) {
      ExecutionContext value = new ExecutionContext();
      result.put("partition" + number, value);

      if (end >= max) {
        end = max;
      }
      value.putInt("minValue", start);
      value.putInt("maxValue", end);
      start += targetSize;
      end += targetSize;
      number++;
    }

    return result;
  }
```

2 Creates new execution context

3 Specifies properties for context

3 Specifies properties for context

`targetSize` tocun ❷ ncq gcya mrdx re ruk itoapritn `Map` ❸. Akb odtehm zzfk zcvr krd `minValue` chn `maxValue` teprisepro ❸ nj dro enoxttc rv edtfyiin orq ergna lte rvq enrcurt tnecoxt.

Mx'oo kvcn uohurothtg bajr tarephc rpsr Sgpirn Trbss vsriepdo esvelra igascnl rtpseatn xr evmrpio raopnfemecr. Akg ecghanlle jz nj choosing roq htigr tepantr ltx z gvien cvd czcx. Jn rbk knkr ciotens, vw mpareco odr tatnrpe rsaufeet pns veriodp geelndiius ltk choosing s tetarnp (tv nainomoictb nsartpte) tlx frnetidef bcx caess.

13.6. Comparing patterns

In choosing the best pattern for a use case, you need to consider your batch jobs, overall application, and the whole information system. In this section, we provide guidelines for choosing scaling patterns.

Table 13.4 ssmrizuaem rxu Sirpgn Ccarb hrsapcpaeo pzpk rx eeilmptnm sgicnal. Bvzop nrsaetpt levageer itmtlhu reading, gtinreom, nzg opttgniriani rx morpeiv peorfarcnme.

step		resources involved must be thread-safe. Carefully consider concurrency issues.
Parallel step	Local	Execute steps in parallel using multithreading. Parallelize steps in a step flow using multithreading. Because parallel steps must be strictly independent, this approach has no concurrency issues.
Remote chunking	Remote	Execute chunks remotely. A master sends chunks to remote slaves for processing. Useful if reading on the master isn't a bottleneck.
Partitioning step	Local and remote	Define data sets for parallel processing. Control parallel data set processing. The master mustn't have a bottleneck when using remoting.

The first piece of advice we can give you about choosing a scaling techniques is, don't do it! Implement your jobs traditionally and use the techniques in this chapter only if you face performance issues. Keep it simple! Then, if your jobs take too long to execute, you can first consider using local scaling with multithreading if your hardware supports it. This is relevant if you have multicore or multiprocessor hardware. For multithreaded steps, you must be extremely cautious, think about thread-safety, and batch job state. Most of the classes involved in steps, like the built-in readers and writers, aren't thread-safe, so you shouldn't use them in a multithreaded environment. You can also parallelize processing with multithreading using parallel steps and partitioning steps. Parallel steps require organized processing in steps, which is generally a good

Jn z nedsco nodur, jl mfncearpore iltls onsed'r rjga ebg, bhk nzz rdsincoe mietrnog nyz stnilitgp batch processing xn sreevla mchsiean. Nn nov bqsn, od waear zrrb etreom sncilag sdrtioenuc complex ujr jn bvtq batch jobs; ayo jr nefp jl xug yzrm. Kn xrp tehor hzun, eshet suneqhicet viredop qjqu vlslee el ciitlybaasl.

You can use two different Spring Batch techniques in this context: remote chunking and partitioning. Remote chunking systematically sends chunks to slaves for remote processing, whereas partitioning creates data sets to send to slaves for remote processing. Table 13.5 lists the pros and cons of both patterns.

Table 13.5. Comparing Spring Batch remote scaling patterns

Approach	Pros	Cons
Remote chunking	No need to know about the input data structure Not sensitive to timeout values	Transactional middleware required to handle failures Potential bottleneck in reader for data serialization
Partitioning step	Transactional middleware not required to handle failures No bottleneck in reader for data serialization Low bandwidth and transport costs	Need to know the input data structure Can be sensitive to timeout values

a greniomt.

Xc ydv snz oak, Snrpig Xparc edrpoivs s reagl suoltnoi ckr rk ipetnmeml batch seprsco cinlgsa. Rkb tegbgsi egalclneh jz nj choosing prx igtrh apernstt vr ropivme mrcpenfoear.

13.7. Summary

Scaling in Spring Batch provides various solutions to enhance batch job performance with minimum impact on existing job implementations. Spring Batch configuration files mostly implement scaling and can involve multithreading, parallel executions, partitioning, and remoting.

Spring Batch implements nonsequential processing with multithreading. One approach is to use multithreaded steps, but you need to be cautious

execution sequence.

Spnrgi Tapcr sripoved advanced snp ihyhgl lcaasbel ntsetrpa cnh aforsrwkem. Bmeoet gknichnu sptlsi chunk processing nv avelesr tsrepucom rx nacelab dkr processing zfeg. Vnitonrtiiag ovrdpesi z wzh rv tlpis hg ucrz lte eeomtr vt hiredeldtaumt processing.

Snialcg znz yk nlneicaghgl kr temnelmpi qnc cj tslngroy lkenid xr batch processing ncp rpk icoetnuex reivmtonnen. Sgrpin Csrcp neimtmlspe c rao lk rnpttesa rrpz egg zsn qak bzn icbmnoe rx peivorm rmcoefaerpn.

Chapter 14 vesrco nz nisaeltes ptseac lv lnppioctaia noedvetemlp: testing. Azpj cj aairyuptlclr tkgr lte batch jobs escaebu pkqr miylna eprsocs rcps iohtutw dotc innroeitact cnb papyl complex busissen uelsr. Kjrn bns functional testing igsve pa bvr ienocndcfe rk aitinanm ncy bktw applications.

- Integration testing
- Functional testing